Metacomputing and Resource Allocation on the World Wide Web

by

Mehmet Karaul

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 1998

Approved: ____________________

Zvi M. Kedem

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **MAY 1998**	2. REPORT TYPE	3. DATES COVERED -
4. TITLE AND SUBTITLE **Metacomputing and Resource Allocation on the World Wide Web**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT **see report**		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**142**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

To Helen and Zachary

Acknowledgements

I want to thank Zvi for being a great advisor. He always puts his students first, and he has given me all the support I needed throughout my studies. I am grateful to him for teaching me to distinguish between important details I should pay attention to and unnecessary cludder which I should dispose of; he pushed the right buttons to make me realize this over and over. I hope he will remain my advisor, even after I graduate.

And I want thank Yannis Korilis for giving me the opportunity to work with him on a great project. An important part of this dissertation represents the work we accomplished together at Bell Laboratories. I have no doubt that we will continue working together on this and other projects in future.

Others deserving my thanks for various reasons include: Arash Baratloo, Partha Dasgupta, Arthur Goldberg, Ayal Itzkovitz, Holger Karl, Ariel Orda, Ken Perlin, and Peter Wyckoff.

Then, there are the most important people in my life. My wife, Helen, who

put up with my working through numerous nights and weekends with remarkable patience and support, especially the last few months. And my son, Zachary, who, I hope, will forgive me for missing out on so many days of his young life—once he is old enough to understand.

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; and by the National Science Foundation under grant number CCR-94-11590. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

Contents

Dedication **iv**

Acknowledgements **v**

List of Figures **xi**

1 Introduction **1**

1.1 Motivation 1

1.2 Resource Allocation 2

1.3 Metacomputing on the Web 5

1.4 Network Computing with Java Applets 7

1.5 Outline of the Dissertation 9

2 Resource Allocation **10**

2.1 Introduction 10

2.2 Design 14

2.3 Name Resolution 16

2.3.1 Replica Address Cache 16

2.3.2 Propagation of Addresses 19

2.4 Routing Strategies 21

2.5 Load Distribution 24

2.5.1 Pricing Strategies 26

2.5.2 Adaptive Algorithm to Determine Weight Factors 27

2.6 Implementation 30

2.7 Experiments 34

2.7.1 Minimizing End-To-End Delays 34

2.7.2 Dynamic Performance Changes 36

2.7.3 Changes in Server Pool 38

2.7.4 Enforcing a Target Load Distribution 38

2.8 Related Work 44

2.9 Conclusions 46

3 Metacomputing on the Web **48**

3.1 Introduction 48

3.2 Virtual Machine 53

3.2.1 Programming Model 55

3.2.2 Distributed Shared Memory Semantics 56

3.3 Syntax and Semantics of Charlotte Programs 58

3.4 Eager Scheduling and TIES 63

3.5 Runtime System 65

3.5.1 An Overview of the Execution 67

3.5.2 A Sample Execution 70

3.6 Matching Workers with Computations 74

3.7 Distributed Shared Memory 77

3.7.1 Existing Techniques 77

3.7.2 Implementation 79

3.8 Experiments 82

3.8.1 Setup 82

3.8.2 Results 86

3.9 Related Work 90

3.10 Conclusion 93

4 Network Computing with Java Applets **94**

4.1 Introduction 94

4.2 Related Work 97

4.2.1 Parallel Computing 97

4.2.2 Collaborative Applications 99

4.3 Design 100

4.3.1 Registration and Lookup 102

4.3.2 Arbitrary Origin of Applets 105

4.4 Directory Service 105

4.4.1 Registration 107

4.4.2 Lookup 108

4.4.3 Directory Service and Java RMI 111

4.5 Class Server 112

4.5.1 Implementation 113

4.5.2 KnittingFactory and Charlotte 114

4.6 Conclusions 115

5 Conclusions **117**

Bibliography **119**

List of Figures

2.1 One-to-many mapping from hostname to IP numbers. 16

2.2 A distributed Web site with logical address and IP numbers. . . . 17

2.3 HTTP Request message with `Timestamp` header. 18

2.4 HTTP Response message with `Timestamp`/`Addresses` headers. . . 19

2.5 Extended mapping with port numbers and relative path offsets. . . 20

2.6 WebSeAl client agent and server agent. 29

2.7 Request distribution in a dynamically changing environment. . . . 37

2.8 Price and load changes with $\theta = 0.25/f^*$. 39

2.9 Price and load changes with $\theta = 0.5/f^*$. 40

2.10 Price and load changes with $\theta = 1.0/f^*$. 41

2.11 Price and load changes with $\theta = 2.0/f^*$. 42

2.12 Price and load changes with $\theta = 4.0/f^*$. 43

3.1 Perfect virtual machine vs. unreliable execution environment. . . . 54

3.2 Charlotte's execution model. 56

3.3 Matrix multiplication in Charlotte. 61

3.4 Charlotte matrix multiplication code: MatrixMult.java. 62

3.5 Charlotte's runtime environment. 66

3.6 Eager scheduling example. 68

3.7 Sample list of active Charlotte programs. 75

3.8 Distributed objects in Charlotte. 80

3.9 Profiles of machines available for the experiments. 84

3.10 Scalability experiment of a Charlotte Ising model program. 87

3.11 Load balancing experiment of a Charlotte Ising model program. . . 89

4.1 Architecture of typical Web-based parallel/collaborative systems. . 98

4.2 Layered design of KnittingFactory. 101

4.3 The main page of a KnittingFactory registry. 106

Chapter 1

Introduction

1.1 Motivation

Load balancing, fault-tolerance, and scalability play an important role in distributed computing. They are of interest for many distributed systems problems as well as for many problems in other areas. Collaborative applications, for example, need to provide some level of fault tolerance to allow for failures of participants' machines. In networking, as another example, techniques are needed to balance the load offered to network links and to tolerate machine failures. In this dissertation, I will present three projects with different goals to which all or some of these issues are relevant, and present different ways to address them.

My main focus will be on the World Wide Web, but the techniques presented here are not restricted to it. The Web can be viewed as a large, unreliable, distributed platform composed of different machines with different speeds which may slow down, speed up, and crash-fail at any time, and networks with different capacities which may get congested and partitioned. All these factors make the Web a very challenging environment.

The projects I present range from resource allocation, to parallel computing, to infrastructure support for collaborative applications. Following are summaries of these projects.

1.2 Resource Allocation

With the rapid growth of the Web, clients attempting to access some popular Web sites are experiencing slow response times due to server load and network congestion. Many sites have replaced the single server with a cluster of replicated servers to partition server load. To address the issue of network congestion, some sites have chosen to geographically disperse the replicated servers—this approach has become popular with software archives which have *mirror* sites, typically on several continents. Such a distributed architecture may result in increased availability of the service in times of network congestion and partial unavailability, and increase performance by taking advantage of "proximity" between clients and servers.

Distributed Web sites require means to control the load distribution over the replica pool. Designing an allocation strategy for a distributed Web site which does not sacrifice any of its benefits is a challenging task. Any successful solution must address the following issues: geographic distribution, scalability, load balancing, fault transparency, dynamic changes in the server pool, transparent name resolving, flexibility, and legacy code and standards. To address these issues, I designed and implemented WebSeAl [34].

Most existing approaches can be classified into two categories: (1) DNS based approaches which generally suffer from load imbalance due to caching of mappings, and (2) server side approaches which do not address geographic distribution appropriately. WebSeAl is closer in nature to the work presented in [63]. Unlike traditional approaches, it pushes the server allocation functionality onto the clients. A client agent module located at the client side and a server agent module at each replicated server implement WebSeAl's functionality. This functionality should ideally be included in Web servers and browsers/proxies. The current WebSeAl prototype provides a client agent and a server agent, realized as separate processes, to implement the required functionality without requiring any changes to existing software.

Each client maintains a cache of logical hostnames and IP numbers to perform one-to-many mappings; servers include this address information in their responses. To create an initial entry in its cache, the client uses standard DNS name resolving and sends the request to the corresponding server. To keep its cache up-to-date, the client includes a timestamp in its requests which indicates the state of the currently cached mapping for any given distributed server; servers inspect this timestamp and include in the response address updates only.

The client strives to minimize the end-to-end delay of each HTTP request. It intercepts each request, makes routing decisions based on the response times of each replicated server, and forwards the request to the selected server. It uses the delays of previous messages to compute estimates and avoids any *probes* or control messages. WebSeAl's routing strategy favors the more responsive servers while using the slower ones occasionally to detect potentially improved servers.

WebSeAl uses pricing mechanisms to provide means for service providers to effectively control the utilization of individual servers. Building on theoretical work based on game theory, it uses cost functions to make the distributed clients implement routing strategies which lead to any desired load distribution. For this, clients base their routing decisions not only on performance statistics, but also on the service cost for each server. The underlying methodology is motivated by recent analytical studies in the area of networking which have shown that a network/service provider can enforce any desired load distribution by means of appropriate pricing strategies even if clients make their routing decisions noncooperatively [44, 45].

I believe pushing the routing functionality onto the clients scales well and results in increased performance in many cases. I have conducted a series of experiments using Web servers on six continents and the results show that WebSeAl can deliver significant performance gains while imposing little overhead [34].

This is joint work with Yannis A. Korilis and Ariel Orda.

1.3 Metacomputing on the Web

The utilization of local area networks as a parallel computing platform has been popular for many years and several research projects such as PVM [57], MPI [28], and Calypso [4] have aimed at this goal. Based on their success, and as an evolutionary step, attempts have been made to extend existing approaches from local area networks to wide area networks such as the Web, with the promise of tapping into a larger resource of mostly idle machines. The Web, in conjunction with Java, has greatly increased its potential of being used as an inexpensive and convenient metacomputing resource, making it an ideal candidate for compute-intensive applications.

Utilizing the Web as a metacomputing resource introduces new challenges different from those that exist in local area networks. The Web invalidates many of the assumptions made to build parallel environments for workstation clusters. For example, there is no shared file system, no one has accounts on all connected machines, and it is not a homogeneous system. To effectively utilize the Web as a metacomputing resource, one needs to address various important issues such as programmability, heterogeneity, portability, security, dynamic execution environment, and scalability. My work on Charlotte [7] addresses these issues by offering a unified programming and execution environment for the Web. Its main contributions are:

- Charlotte is the first environment that allows *any user on the Web, using any Java-capable browser to participate in a parallel computation*, simply by clicking on a link. It leverages the ability of browsers to download and execute remote applets to overcome the need for a shared file system or remote accounts.

- A new technique for providing a *distributed shared memory abstraction without relying on operating system or compiler support.* It implements a shared name space at the language level without requiring operating system support (not possible with Java) or modifying compilers or the JVM (not desirable).

- Charlotte's runtime system provides *load balancing and fault tolerance* to deal with the dynamics of the Web by using two integrated techniques, *eager scheduling* and *two-phase idempotent execution strategy*.

- Charlotte *decouples the programming model from the execution environment.* Its programming environment can conceptually be divided into (1) a virtual machine model which provides a reliable shared memory machine abstraction to the programmer and isolates the program from the execution environment, and (2) a runtime system which realizes this model on a set of unpredictable, dynamically changing, and faulty machines.

- Charlotte is the *first programming environment for parallel computing using a secure language.* It is built on top of Java without any native code and provides the same level of security as Java.

There have been several projects that focus on Java-based parallel computing. Unfortunately, most fail to take advantage of Web browsers' ability to download and execute applets in a secure fashion. This is a crucial requirement if a system is to entice users to donate their resources. Charlotte was the first system to achieve this. It was followed by other similar systems like Javelin [14], Bayanihan [54], and Ninflet [58], strengthening Charlotte's original argument that the Web is an attractive metacomputing resource.

This is joint work with Arash Baratloo, Zvi M. Kedem, and Peter Wyckoff.

1.4 Network Computing with Java Applets

While working with Charlotte, some of its limitations became clear. A deeper look revealed that other Web-based parallel computing environments and many collaborative applications, such as shared whiteboards and editors, have similar shortcomings. My work on KnittingFactory [6, 5] addresses these issues and provides infrastructure support to build such applications without the aforementioned limitations.

KnittingFactory focuses on applications composed of Java applets and designed to run within Web browsers. As a result of restrictions put on applets, specifically the *host-of-origin* policy, the architectures of many existing systems have the following characteristics: (1) a stand-alone application running on the same host as an HTTP server; and (2) users on the Internet joining a stand-alone application by loading an applet using an a priori known URL and executing it within a browser. In an ideal situation, (1) the application should not be tied to an external HTTP server and should be allowed to execute anywhere on a network; and (2) users should not be required to have a priori knowledge of the URL and should be able to locate these applications in a seamless fashion via browsers. KnittingFactory is an infrastructure for building such systems. It provides two integrated services:

- *KF Directory Service*, a distributed name service to assist users in finding networked applications on unknown hosts. It allows users to find such applications on the Internet by performing a lookup at any site offering a *KF Directory Service*. It works fully within browsers and uses JavaScript.

- *KF Class Server*, an embedded class server to eliminate the need of external HTTP servers. It provides a light-weight solution to allow users to initiate a distributed effort on any host.

This is joint work with Arash Baratloo, Holger Karl, and Zvi M. Kedem.

1.5 Outline of the Dissertation

This dissertation is organized as follows: I will first present WebSeAl and discuss how it addresses scalability, load balancing, and fault masking in Chapter 2. Next in Chapter 3, I will describe Charlotte and the techniques it uses to deal with machines of different speeds and crash-failed machines. In Chapter 4, I will present an infrastructure Web-based applications which contains various aspects relating to scalability and fault masking.

Chapter 2

Resource Allocation

2.1 Introduction

The rapid growth of the World Wide Web has led to a steady increase of client requests to many popular Web sites. Both, overloaded servers and network congestion contribute to slow response times of such sites. It may not be cost-effective to upgrade the server machine with a more powerful one, especially when incremental scalability is desired. Instead, most sites opt to replace the single server with a cluster of replicated servers [50, 51]. Although this may solve the problem of overloaded servers, it does not address network congestion. Some sites choose to geographically distribute the replicated servers—this approach has become popular with software archives (e.g. [61]) which have *mirror* sites, typically on several continents. Such a distributed architecture may result in increased availability of the service in times of network congestion and partial unavailability, and it may increase performance by taking advantage of proximity between clients and servers. Typically, the service content is replicated or a distributed file system [30, 55] is used.

Service providers using such a distributed architecture require means to effectively control the usage of individual replicas. More specifically, they require means to route[1] requests in a way such that the resulting *load distribution* coincides with the target load distribution the service provider seeks. A target load distribution might be to partition the incoming requests evenly among the replicas; or the service provider might want to temporarily discourage clients from using a certain replica, for maintenance reasons or to perform other site specific tasks, for example.

Currently, many distributed Web sites require the user to manually select a server out of a list of replicas. This is inconvenient for the user, and a decision to use a certain server might result in poor performance depending on network conditions and the load of the selected server. Furthermore, this approach does not allow service providers to control the load distribution among the replicas.

Designing a transparent allocation strategy for a distributed Web site which does not sacrifice any of the benefits of such a distributed architecture is a chal-

[1]Throughout this Chapter, the term *routing* refers to the assignment of requests to replicas and is not to be confused with routing within the network.

lenging task. A successful solution must meet several requirements:

- **Load Balancing:** Service providers should be able to effectively control the utilization of individual servers.
- **Geographic Distribution:** Network delays between a client and individual servers of a distributed service might differ significantly. Server allocation should take advantage of this while still accommodating dynamic changes in network performance and server load.
- **Scalability:** Server allocation should gracefully scale with the increasing number of clients.
- **Transparent Name Resolving:** Popular Web sites have well publicized server names and require a transparent mapping to replicated servers.
- **Dynamic Changes in Server Pool:** Addition, removal, and migration of servers should be supported, and changes should be reflected as quickly as possible.
- **Fault Transparency:** Unresponsive machines should be detected and requests transparently redirected to other replicas. Also, previously unresponsible machines which become available again should be incorporated quickly.
- **Flexibility:** Different users may have different objectives when accessing Web sites, requiring support for customized strategies.

- **Legacy Code and Standards:** It should not require any changes to existing client or server code and should conform to existing standards.

A comprehensive solution for allocation of distributed Web servers must address all these factors. I am not aware of any system which achieves this. In this Chapter, I present a system called *WebSeAl* which addresses these issues.

The research leading to WebSeAl is based on theoretical work where provable methods for controlling network load using pricing mechanisms were developed [44, 45]. It was shown that even with noncooperative clients (in a fully distributed, and therefore scalable fashion), the network load can be controlled effectively. The work presented here applies these techniques to provide scalable and controllable load balancing for distributed Web servers.

The remainder of this Chapter is structured as follows. I will first present the decentralized design of WebSeAl and then describe how logical names are resolved. Sections 2.4 and 2.5 discuss how WebSeAl can enforce any desired load distribution over the server pool. In Section 2.6, implementation issues regarding the prototype I designed and implemented are addressed. This will be followed by experiments and concluding remarks.

2.2 Design

A simple solution to enforce a load distribution over a set of replicated servers is to make sure that all requests pass through some sort of a central dispatcher, which directs individual requests to different replicas depending on the target load distribution. Since this central dispatcher has global knowledge, ensuring a specific load distribution is not difficult. However, such an approach raises scalability concerns. Also, it does not address the issue of geographic distribution appropriately; a client will not be able to take advantage of a close-by replica if the central dispatcher is located far away since each request needs to pass through the dispatcher.

WebSeAl uses a decentralized approach to address the issue of scalability and geographic distribution. In WebSeAl, it is actually the client who makes routing decisions. At times, I will refer to such clients as noncooperative. There is no central entity which dispatches the requests. This fully distributed approach should scale well with the number of clients, and it makes it possible for clients to take advantage of geographic proximity, specifically, fast network connections between the client and certain replicas.

The basic functionality at the client side can be described as follows:

- Clients make routing decisions.
- They maintain a cache of address information.

- They collect dynamic performance data (e.g. network conditions, server load, and other site specific data).
- Clients base routing decisions on this data.
- They automatically redirect the request to an alternate server if the selected server is not responsive.

Besides delivering the actual service, servers provide clients with address information. They also communicate other site specific data which might be used to control access to the server pool, to support charging for services, and so forth, as will be discussed in Section 2.5.

With such a decentralized approach, the actual load distribution among the server pool will solely depend on how clients make their routing decisions. Therefore, the challenge is to make sure that clients implement routing strategies so that the resulting load distribution coincides with the desired one. This is a much harder problem compared to the centralized approach since each client has only "partial knowledge", and there is no entity which has global knowledge. I will discuss in Sections 2.4 and 2.5 how WebSeAl ensures that any target load distribution can be enforced. Before addressing these issues, I will first describe how clients find out about replicas.

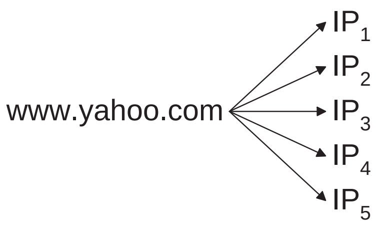


Figure 2.1: One-to-many mapping from hostname to IP numbers.

2.3 Name Resolution

A server is identified by a logical address in the form of a hostname. When a client attempts to contact a server, the Domain Name Server (DNS) system transparently resolves the hostname to an IP number, which is successively used to establish the connection. To contact a distributed server in a transparent fashion, a *one-to-many mapping* from the hostname to one of the IP numbers of the replicated machines is needed. WebSeAl pushes this name resolving functionality onto the clients.

2.3.1 Replica Address Cache

As mentioned earlier, clients maintain a cache of replica address information. They use this information to access a distributed Web site identified by a hostname, e.g. `www.yahoo.com`. A mapping consists of the hostname and the IP numbers of the replicas making up the distributed service. Using this information, clients perform a one-to-many mapping from the public hostname to the IP numbers of the individual replicas (see Figure 2.1).

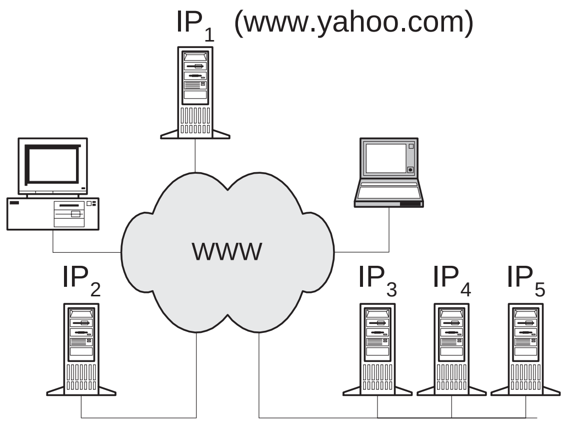


Figure 2.2: A distributed Web site with logical address and IP numbers.

When a client attempts to access a distributed server for which it does not have a mapping cached, it uses standard DNS name resolving and contacts the server at that hostname. This means that one of the replicas is known to the standard DNS system by the logical address of the distributed Web site. More specifically, a distributed Web site consists of a set of servers $S_1, \ldots, S_n$, each with its own IP number $\mathrm{IP_1} \ldots \mathrm{IP_n}$ (Figure 2.2). One of these servers is known to the standard DNS system by the hostname of the distributed Web site. This server will successively communicate the addresses of all replicas to the client (as will be discussed below). Future requests to this distributed service use this information to perform one-to-many mappings from the logical address to the individual hosts. The standard DNS system is used only for bootstrapping—once a mapping for a logical address is cached, the DNS system is not needed to access any of the replicas. The service

```
GET http://www.yahoo.com/index.html HTTP/1.0

Timestamp:  Sun, 06 Nov 1994 08:49:37 GMT

...
```

Figure 2.3: HTTP Request message with `Timestamp` header.

will be accessible as long as *at least one* replica remains responsive.

Servers communicate address information to the clients. Each server is assumed to have address information about all replicas—this information can be made available to individual replicas in the same way the service content is made available to them. Servers include the addresses of the individual servers in the actual response they generate. Clients extract these addresses from the response and update their cache accordingly.

Clients need to retrieve the addresses of the servers only to create an initial entry or to refresh their cache if the address information has changed in any way. To avoid unnecessary transmission of address information, clients include a timestamp in their requests which indicates the state of the currently cached mapping for the given distributed server. This timestamp is provided by servers along with the addresses. Upon receipt of a request, a server inspects this timestamp and includes the addresses and the new timestamp in the response only if more up-to-date address information is available. This is very similar in nature to the

```
HTTP/1.0 200 OK

Timestamp:  Sun, 06 Nov 1994 08:49:37 GMT

Addresses:  IP1 IP2 IP3 ...

...
```

Figure 2.4: HTTP Response message with `Timestamp`/`Addresses` headers.

`If-modified-since` header [9], which is used to avoid retrieving cached files which have not been modified since a certain date.

2.3.2 Propagation of Addresses

HTTP allows application specific header fields and requires that all intermediaries such as proxies or gateways conforming to HTTP ignore these and forward them unchanged. This is used to piggyback timestamps and addresses in HTTP messages. WebSeAl introduces two new message headers: `Timestamp` and `Addresses`. Clients use the first header to notify servers about the status of their cached addresses for the distributed server at hand (see Figure 2.3). Servers use both headers to return a list of addresses and the timestamp at which this information was generated (see Figure 2.4).

The format of a `Timestamp` header is defined as follows:

`Timestamp:` *HTTP-date*

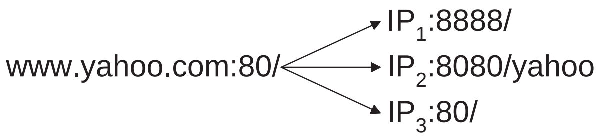


Figure 2.5: Extended mapping with port numbers and relative path offsets.

`HTTP-date` is the standard date/time stamp format used on the Internet as defined in [9].

The format of a `Addresses` header is defined as follows:

`Addresses:` *IP-number-list*

`IP-number-list` is the list of IP numbers separated by whitespaces.

Mapping a hostname to a set of IP numbers shares many similarities with DNS based and server side approaches which will be described in Section 2.8. Notice that these approaches require that the servers on all replicated hosts accept connections at the same port. In addition, the directory structure must be identical on each host. WebSeAl's architecture relaxes these restrictions. The mapping from hostname to IP numbers can easily be extended to a mapping from hostname and port to IP number and port to accommodate the usage of different port numbers. This requires that the address information included in responses be extended to contain port numbers as well as hostnames. Path offsets can be accommodated similarly (see Figure 2.5). For example, `www.yahoo.com:80/` can be mapped to `122.140.128.40:8080/yahoo/`. On the first host, the server is accepting connections at port 80 and the directory structure is rooted at `/`. On the second host, the server accepts connections at port 8080 and the root directory is at `/yahoo/`. Many mirror sites use different root directories and require a relative path offset. This allows a single host to serve as a replica for multiple distributed Web sites.

2.4 Routing Strategies

As mentioned before, in WebSeAl it is the clients who make routing decisions. When clients make routing decisions, the resulting load distribution over the server pool will be solely the result of the routing strategies the clients implement. To be able to control the resulting load distribution, mechanisms are needed to influence the client side routing decisions. We use a pricing scheme to achieve this in WebSeAl. I will present the details in two steps: In this Section, I will assume that the service provider does not attempt to enforce a specific load distribution, and that the only objective for clients is to minimize their own delay. In Section 2.5, I will extend these techniques and present how WebSeAl can enforce any target load distribution.

The fact that many Web pages contain several images and frames results in the generation of several requests to retrieve a single Web page. WebSeAl clients measure the total response time for each such request. The total response time measured is the complete end-to-end delay which includes connection establishment, network delay, and server time. WebSeAl clients strive to minimize this total delay.

Each client makes routing decisions based on the average response time of each server. These averages are estimated using the measured response times for the N most recent requests, for some N. The updated routing strategy is used to direct the next N requests to the appropriate servers in the pool. Alternatively, the client could estimate the average response times by sending occasional *probes* at the cost of increased network traffic. In the current prototype, we decided against this approach to avoid control messages between clients and servers.

One possible routing strategy clients could employ is to always contact the most responsive server. This approach, however, will fail to collect new performance data for the slower servers. Instead, we use *probabilistic routing* to ensure that clients collect new performance data for all servers. More specifically, if T_i denotes the average response time for requests routed from a client to server i, then the client will route its next N requests based on the probability distribution:

$$p_i = \frac{1/T_i^k}{\sum_j 1/T_j^k}, \tag{2.1}$$

where the exponent $k \geq 0$ is a constant.

With $k = 0$, requests are routed to the servers randomly, without taking into account their performance. With $k = 1$, we can achieve linear distribution. This will favor fast machines while still using slower ones. However, the overall performance might suffer due to possibly long delays from slow servers. By raising k, more requests will be routed to the most responsive servers.[2] Very high routing probabilities for the fastest servers will cause very infrequent usage of slower ones, which in turn will decrease the potential to quickly detect improved servers. WebSeAl imposes a minimum threshold to prevent very low probabilities.

In the current prototype, routing decisions are based on the most recent N measurements. We are considering several alternate strategies:

- **Sliding Window:** Instead of calculating estimates every N requests using the last N measurements, one could update the estimates after every measurement, always using the last N measurements.

- **Weighted Average:** When calculating the performance estimates of replicas, more recent data should impact the overall performance more than older data, and the estimates should be updated more frequently.

- **Time-of-Day:** Network conditions and server usage vary with the time-of-day or the day of the week [17, 32], and this information could be considered in the routing strategy.

WebSeAl allows different clients to use different routing strategies. As future work, we plan to experiment with various strategies and to investigate how each

[2]When k is (very) large, all requests will be routed to the most responsive replicas.

one and various combinations perform in different settings. Our goal is to realize a set of routing strategies and to adapt dynamically to changing conditions.

2.5 Load Distribution

Using the results from the previous Section, the load distribution over the server pool (also called *operating point*) is solely the result of the interaction among the distributed clients and cannot be controlled by the service provider. In this Section, I will discuss strategies which can be used at the server side to control the load distribution while clients make their routing decisions in a noncooperative manner.

The service provider aims at distributing the load currently offered to the server pool in a way that is deemed efficient from the *system's* point of view. The provider, for instance, might desire a load distribution which minimizes the *overall* average response time of the server pool. In other cases, the provider might want to discourage usage of certain machines—even if they are the most responsive ones—in order to perform other site specific tasks. Therefore, a mechanism is needed to make the distributed clients implement routing strategies which lead to a load distribution that coincides with the desired one.

The problem of managing the behavior of systems where control is distributed and noncooperative is a fundamental one. The interaction among the various distributed controllers (the clients in WebSeAl) can be modeled as a *game*, and

Game Theory provides the systematic framework to study and analyze the behavior of such systems—for an overview of game theoretic aspects in computer networking see [42] and references therein. The operating points of the system are the *Nash equilibria* of the underlying control game. Noncooperative equilibria are inherently inefficient: while each controller strives to optimize its individual performance, the overall behavior of the system is generically suboptimal.

WebSeAl uses a *pricing mechanism* to provide incentives to the noncooperative clients to implement routing strategies that lead to the desired load distribution over the server pool. The methodology is motivated by recent analytical studies in the area of networking which have shown that a network/service provider can enforce any desired operating point by means of appropriate pricing strategies [44, 45]. The key idea in WebSeAl's pricing mechanism is that there is a *weight factor* associated with obtaining service from each server in the pool. Clients make their routing decisions based not only on performance statistics, but also on weight information for each server. The main assumption behind this mechanism is that the clients are indeed sensitive to weight factors. This behavior is expected in private Intranets where clients and the pricing mechanism are part of the same management system. For external clients accessing a Web site, this behavior can be enforced by actual usage-based service charges (for commercial Web sites), or by means of limited electronic budget allocated to each client—an architecture developed according to these ideas is proposed in [43]. When clients are sensitive to weight factors, the service provider can control not only the load distribution over the available servers, but also the total offered load itself.

2.5.1 Pricing Strategies

The goal of the pricing mechanism in WebSeAl is two-fold:

- Avoidance of congestion (overload conditions) at various servers.
- Load balancing—that is, distribution of the total load offered to the Web site among the available servers in a way that is deemed efficient by the provider.

The pricing strategies in the current version of WebSeAl are based on analytical results in [45]. That study considers a system of general network resources accessed by a number of noncooperative clients. Each resource is characterized by its "capacity," that is, the maximum load that can be accommodated by the resource. *Congestion pricing* is proposed as a means for avoiding overload conditions: the weight factor per size unit (i.e., the price) of each resource is proportional to the congestion level at the resource that depends on the total load offered to it by the clients. More specifically, the price of each resource is given by the congestion function associated with the resource multiplied by a weight factor. These weight factors determine the relative sensitivity of the clients to the congestion level at the various resources. Load balancing can be achieved by appropriate choice of these weight factors. This pricing strategy is shown to allow the provider to enforce any desired operating point while the clients make their routing decisions noncooperatively.

Along the lines of these analytical results, the pricing strategy in the current design of WebSeAl is based on a weight factor for each server in the pool, which determines the relative sensitivity of the clients to the responsiveness of the server.[3] In particular, the performance metric considered by each client in making its routing decisions is the average response time of each server multiplied by the corresponding weight factor. Therefore, if w_i is the weight factor of server i, and T_i the average response time from the server to a client, then the routing strategy of the client described by eq. 2.1 becomes:

$$p_i = \frac{1/(w_i T_i)^k}{\sum_j 1/(w_j T_j)^k}. \tag{2.2}$$

2.5.2 Adaptive Algorithm to Determine Weight Factors

Server weight factors are determined based on the operating point the provider wants to enforce. One way to determine these factors is to map the parameters of the model considered in [45] to the characteristics of WebSeAl and apply the corresponding analytical results, expecting to achieve a good approximation of the desired operating point. Instead, we choose to use an *adaptive algorithm*,

[3]Note that the weight factor of each server is the same for all clients.

also proposed in [45], which does not depend on the details of the underlying analytical model. The algorithm updates the weight factors iteratively, based on the "distance" of the current operating point from the desired one.

If f_i^* denotes the desired load at server i and $f_i(n)$ the actual load offered to the server during the nth iteration, then its weight factor w_i is updated using the following:

$$w_i(n+1) = w_i(n)e^{\theta_i(f_i(n)-f_i^*)}, \tag{2.3}$$

where $\theta_i > 0$ is a constant which determines the rate of change in the weight factor of server i. The idea behind this iterative scheme is that, if the server is currently receiving less load than the desired one, its weight factor should be decreased. This decreases the clients' sensitivity to the congestion level at the server, thus encouraging them to direct more of their requests to it. Similarly, if the server receives more load than the desired one, its weight factor is increased. Under a set of general assumptions guaranteeing that the client population as a total reacts "rationally" to price changes, this iterative scheme was shown in [45] to drive the system to the desired operating point.

In the current implementation of WebSeAl, server load is expressed in requests per unit of time. Considering HTTP requests, we expect that each client generates a large number of requests, each of which will be small to moderate size. Therefore, this is a satisfactory approximation. A more precise load metric would consider

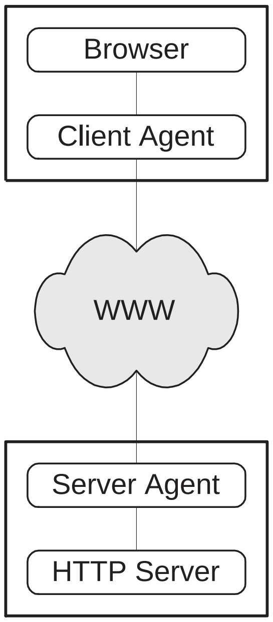


Figure 2.6: WebSeAl client agent and server agent.

the actual size of each request and will be incorporated in future implementations.

Each distributed Web site is equipped with a *pricing manager*. Based on the target operating point, the pricing manager determines the weight factors to access each server and communicates it to the corresponding server. More specifically, the pricing manager periodically collects information about the load offered to each server by contacting the corresponding server, updates the weight factors according to iteration 2.3, and communicates them to the servers. Each server receives only the update of its own weight factor and is responsible for advertising this to the clients. This is achieved by piggybacking the weight factor of the server to HTTP messages which contain the responses to the clients' requests.

2.6 Implementation

WebSeAl's server side functionality could be added to existing Web servers quite easily and should impose only little computational overhead. However, to create a usable system without having to modify existing servers, WebSeAl provides a stand-alone Java application, called *server agent*, which implements the server side functionality (see Figure 2.6). The server agent functionality can be outlined as follows:

- The server agent uses the HTTP port to intercept each incoming request.
- It forwards the request to the local HTTP server.
- It accepts the response
- The server agent adds address information (including timestamps) to the response, as needed.
- It forwards the response to the client.

The client side functionality is somewhat more complex, but it should be fairly straightforward to extend existing Web browsers or proxies to support this functionality. Similar to the server agent, WebSeAl provides a stand-alone Java application, called *client agent*, which realizes the client side functionality in order to provide a usable system without having to modify existing browsers or proxies. We take advantage of the fact that virtually all browsers support proxies to intercept requests. When the client agent is started up, it creates a server socket which accepts HTTP requests, very much like a proxy does. By configuring the browser to use the "proxy" (i.e., WebSeAl client agent), the client agent effectively intercepts each request. More specifically, the client agent functionality can be described as follows:

- A client agent uses the proxy "hook" to intercept each request generated by browsers.
- It maintains a cache of replica address information.
- It makes routing decisions as described above.
- The client agent adds a timestamp to the request.
- It measures the total end-to-end delay of the request.
- It forwards the request to the selected server.
- It accepts the response.
- The client agent transparently redirects the request to alternate servers if a selected server remains unresponsive for a certain timeout period.
- It extracts address information and updates its cache, if necessary.

- It forwards the response to the client.

Proxies are generally used to allow Internet access through firewalls and perform caching of Web documents. WebSeAl's client agent can accommodate proxies in two ways. A client agent can be located between one or more clients and a proxy. Since name resolution is performed at the client agent, the proxy will treat identical documents from different replicas of the same distributed server as different documents and create redundant copies in its cache. Alternatively, the client agent can be located "behind" the proxy. This configuration avoids the problem of redundant copies in the proxy cache. Also, only one address cache and a single set of statistical data is maintained for a number of users, resulting in more up-to-date address caches and more accurate estimates.

Both WebSeAl's server and client agent functionality should ideally be included in Web servers and browser or proxies. We provide client and server agents to enable service providers and users to take advantage of this technology without the need to modify existing systems. Independent of whether agents are used or existing systems are modified, for a system like WebSeAl to gain wide acceptance it needs to be backward compatible with regard to clients and servers lacking this functionality. WebSeAl is backward compatible and supports gradual infiltration:

- **WebSeAl Client and Standard Server:** A standard HTTP server is required to ignore the timestamp header in a request from a WebSeAl client agent and will service the request as usual. The lack of address information in the response indicates to the client agent that it is dealing with a standard server. It can react to this, for example, by infrequently including the timestamp in its future requests in order to update its cache in case this site is upgraded.

- **Standard Client and WebSeAl Server:** A request received by a server agent will not contain a timestamp header if the client lacks WebSeAl functionality. The server can react to this in several ways; two possibilities are: (1) it can service the request in a standard manner without including any address information in its response; (2) it can route the request on behalf of the the client to individual servers.

There are a few issues which are not addressed in the current prototype. All of these seem to be solvable, and we plan to incorporate these in future versions of the system. Two concerns are:

- **Cookies using IP numbers:** Since request may be served by different hosts with different IP numbers, it is possible that the browser stores multiple cookies for the same distributed Web server if the cookies use IP numbers, which can create various problems. One simple solution for this is to use the hostname instead of IP numbers.

- **State at the server and sessions:** Some Web sites use sessions, thereby effectively creating different states among the replicas. If the replicas can be expected to be actively replicated, this does not pose a problem. If not, a simple solution is to assure that a single replica is used throughout a session.

2.7 Experiments

In this Section, I will present performance results. I conducted three series of experiments to test how WebSeAl (1) takes advantage of geographic proximity, (2) dynamically adapts to performance changes, (3) accommodates changes in the server pool, and (4) enforces a target load distribution over a server pool.

2.7.1 Minimizing End-To-End Delays

The first experiment consisted of two tests. I used ten mirror sites of a popular software archive which repeatedly appears in [60] as one of the most accessed Web sites. These tests were conducted under real world conditions, using standard machines, networks, and software. The ten servers were located on six continents: two each in North America, South America, Europe, and Asia, and one each in Africa and Australia. The client was running at New York University. Geographically, the closest server to the client was located in Massachusetts, the second closest in California.

The client running five threads generated 1000 requests for a file of length 4253 bytes. All requests were addressed to a single logical address. A local client agent intercepted each request and provided transparent access to a distributed Web site. Since I experimented with existing Web sites not running WebSeAl's server agent, I added the server addresses manually into the cache of the client. Also, since the servers did not provide the client agent with a service weight factor, the client agent used eq. 2.1 to route requests (the constant k was set to 4.0). The clients only goal was to minimize its own delay, independent of the resulting load distribution at the server side.

The first test consisted of two parts: one using WebSeAl's client agent and one contacting the closest server directly. Using the client agent, the total response time for 1000 requests was 291.6 s. The response time measured is the end-to-end delay which includes connection establishment, network delay, and server time. 95.4% of the requests were serviced by the closest server. The total response time for contacting the closest server directly was 266.9 s. This translates to an overhead of 9.2%. The fact that the WebSeAl client agent sent the vast majority of the requests to the closest server indicates that this server was delivering the best performance. Besides the computational and communication overhead of the client agent, an important factor contributing to this overhead is that 4.6% of the requests were routed to slower servers to update performance data for these machines. As mentioned before, this could be avoided by occasionally sending probes at the cost of generating additional traffic.

In the second test, I used the same setup as above, but ran the test at a different time of the day. This time, only 3.9% of the requests were serviced by the closest site. The total response time was 761.4 s as opposed to 1295.3 s when contacting the closest host directly—an improvement of 41.2%. These two tests indicate that WebSeAl can deliver significant performance gains while imposing only little overhead, compared to the scenario when the user is able to always pick the fastest machine.

2.7.2 Dynamic Performance Changes

The second experiment investigates how WebSeAl client agents adapt to dynamic performance changes of individual servers. The setup was the same as in the previous experiment. As with the previous experiment, the client, using a local client agent, generated 1000 requests to a logical address. After 300 requests, I started downloading several large files from the fastest site, which happened to be the geographically closest one, thus generating additional load at that server. This traffic was discontinued after another 300 requests (see Figure 2.7). Of the first 300 requests, 93.3% were serviced by the closest server. This percentage sank to 11.6% for the next 300 requests, and went up again to 93.2% for the last 400

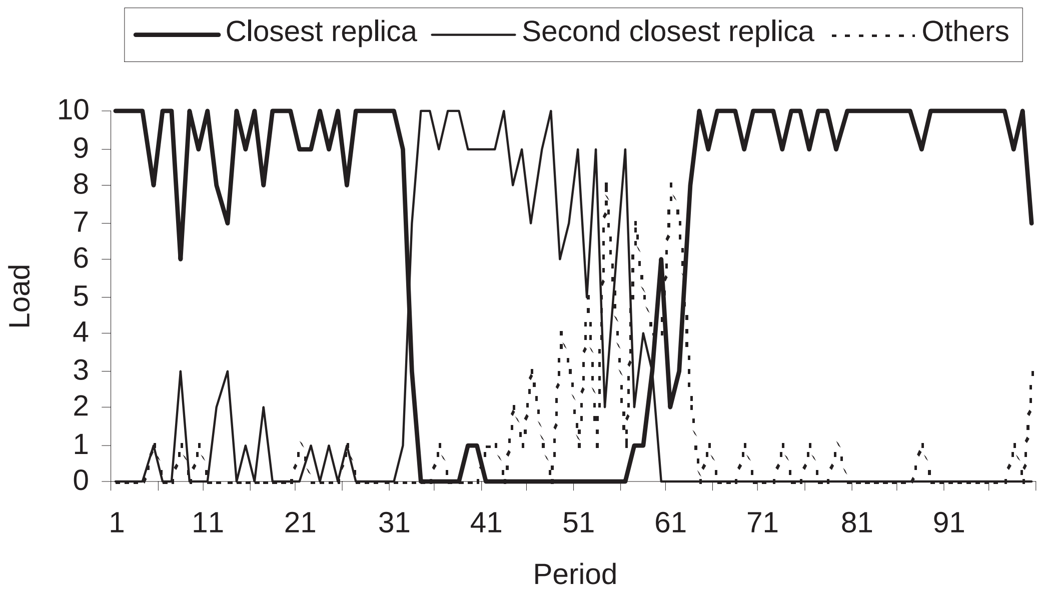


Figure 2.7: Request distribution in a dynamically changing environment.

requests. The second closest server received initially 2.0% of the requests, which increased to 69.3% when the performance of the closest server started to degrade. This indicates that WebSeAl adapts well to performance changes in the server pool.

2.7.3 Changes in Server Pool

For the third experiment, several identical machines in a controlled environment were used to show how WebSeAl reacts to changes in the server pool. On each of four machines, a WebSeAl server agent and a standard HTTP server were started. I first used three servers, added another one after about 300 requests, and removed one of the original three servers after another 400 requests.[4] Since we used identical machines, it can be expected that the two fully available machines would each get 300 requests, the other two each 200 requests. The actual distribution was 295 and 286 requests for the first two machines, and 225 and 194 requests for the other two. This illustrates that WebSeAl quickly and effectively accommodates changes in the server pool.

2.7.4 Enforcing a Target Load Distribution

For the last experiment, I used four identical machines in a controlled environment to show how WebSeAl can enforce a desired load distribution. Two clients were

[4]I removed the server by sending the server agent process a *kill*-signal.

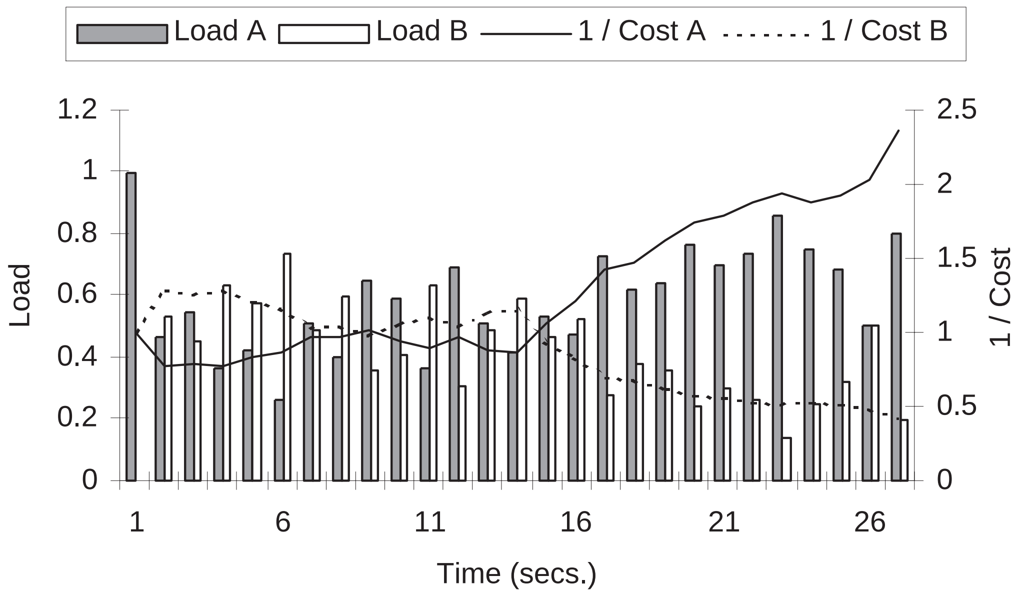


Figure 2.8: Price and load changes with $\theta = 0.25/f^*$.

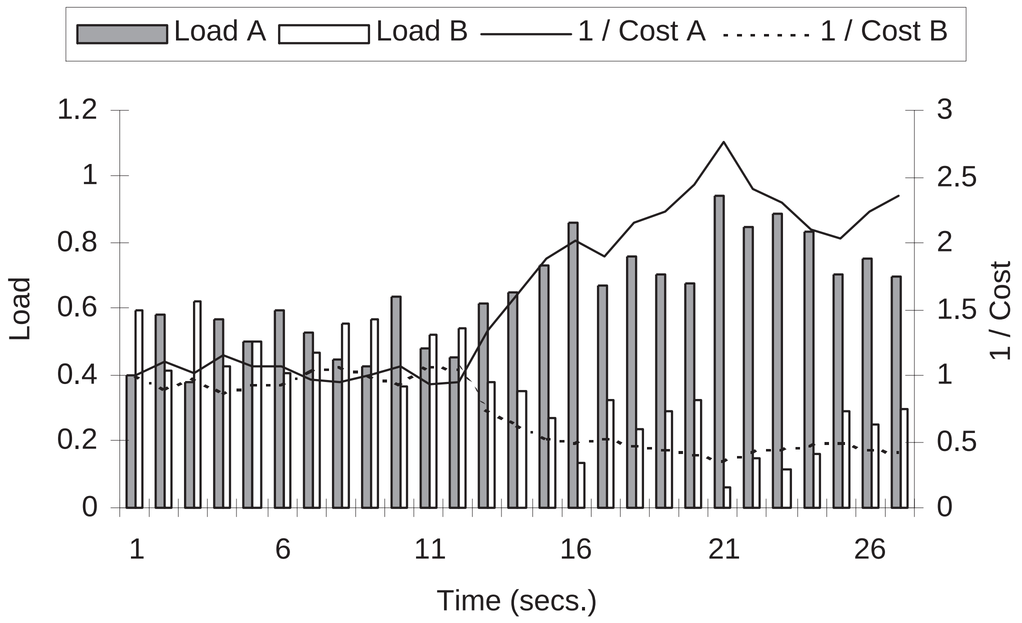


Figure 2.9: Price and load changes with $\theta = 0.5/f^*$.

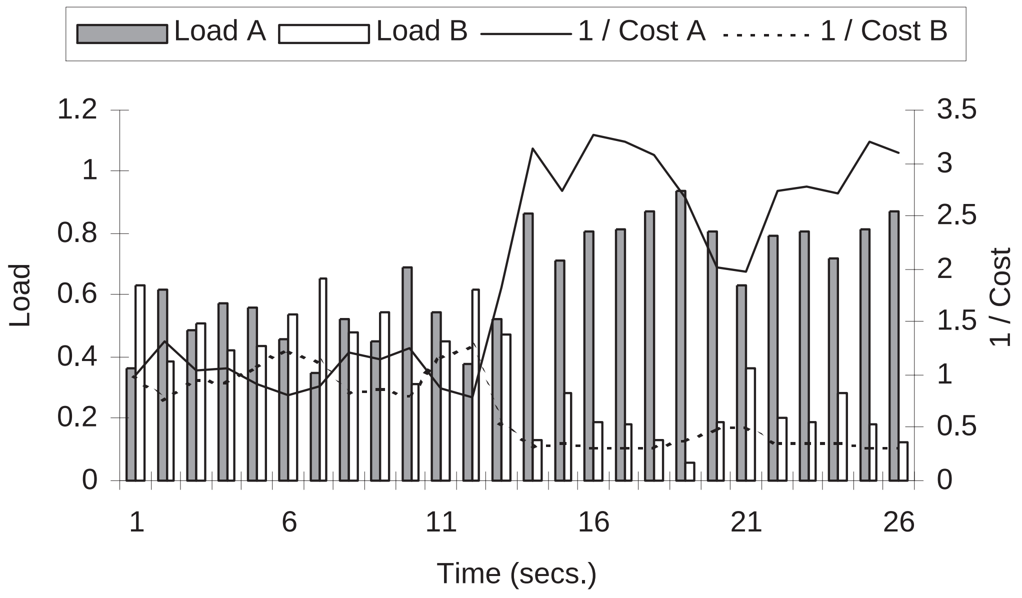


Figure 2.10: Price and load changes with $\theta = 1.0/f^*$.

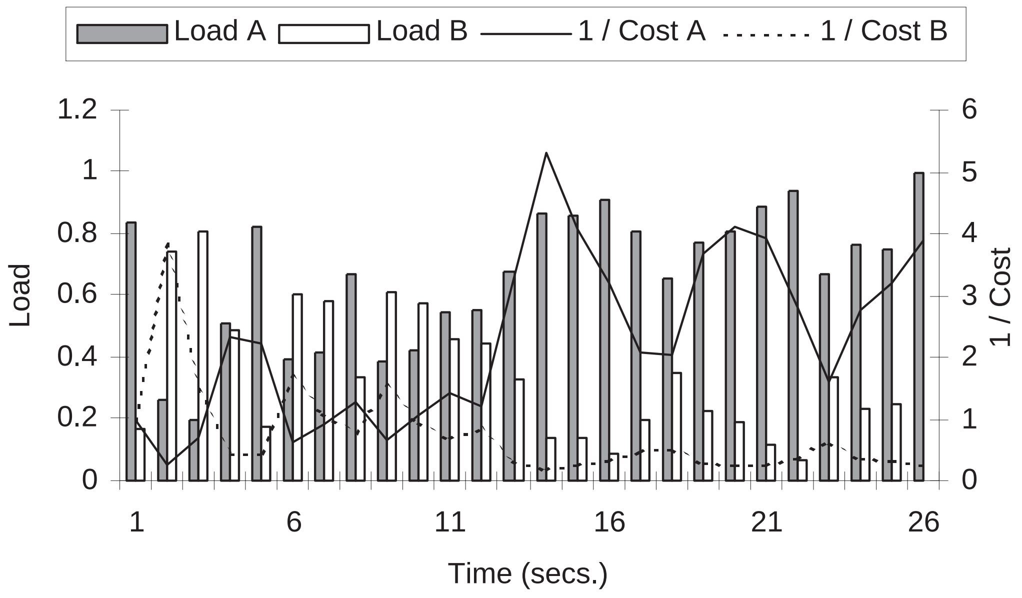


Figure 2.11: Price and load changes with $\theta = 2.0/f^*$.

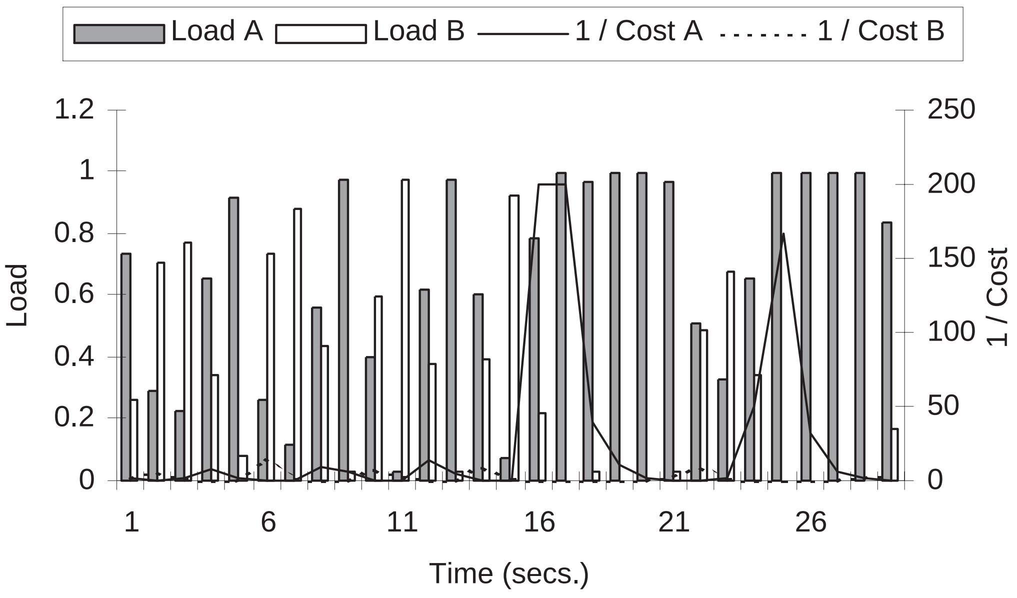


Figure 2.12: Price and load changes with $\theta = 4.0/f^*$.

used, each with a local client agent, and two servers with a local server agent. On a fifth machine, I ran the pricing manager which contacted server agents and updated weight factors every second. Each client generated 500 requests each, for a total of 1000 requests. I started with a target load distribution of 0.5 and 0.5 for the two servers, expecting to distribute the total load evenly. After about 500 requests, I changed the target load distribution to 0.8 and 0.2, expecting to have 80% of the requests routed to one server and the rest to the other one. I ran five tests with θ varying from $0.5/f^*$ to $4.0/f^*$. Figures 2.8,2.9,2.10,2.11, and 2.12, show the prices and the measured load for each server.

As was to be expected, low values for θ result in slow changes to weight factors, leading to slow changes in the measured load. By increasing θ, we can reach a desired distribution faster. However, too high a value for θ results in oscillating prices and load, which is not desirable. More importantly, for the algorithm to converge, θ must be "sufficiently" small [45].

2.8 Related Work

The HTTP redirect [1] approach uses the HTTP return code *URL Redirection* [9] to perform load balancing. A busy server returns the address of another server instead of the actual response, asking the client to resubmit its request to that server. This creates additional network traffic and increased latency. Every request is initially addressed to the publicly known server which creates a single point of failure and the potential for a bottleneck due to servicing redirects.

Domain Name Server (DNS) based approaches [13, 36, 18] perform load balancing at the name resolution level. The name server at the server side is modified to respond to translation requests with the IP numbers of different hosts in a Round-Robin fashion. This results in partitioning client requests among the replicated hosts. The main disadvantage of this approach is that intermediate name servers and clients cache name-to-IP mappings which can result in significant load imbalance. A similar approach is described in [23] where anycast resolvers are used to perform a one-to-many mapping based on locally maintained performance data about individual replicas. The goal is no minimize end-to-end delays, but the issue of enforcing a desired load distribution is not addressed.

Server side approaches [22, 18] use a server side routing module which redirects all incoming requests to a set of clustered hosts based on load characteristics. This is achieved at the IP layer—i.e., the routing module modifies all IP packets before forwarding them to individual hosts. An alternate server side solution which avoids modifying IP packets is presented in [19]. These approaches have the drawback that the routing module represents a single point of failure, and therefore can result in a bottleneck since all requests pass through it. In addition, server side approaches work well only for clustered servers.

Perhaps most closely related to WebSeAl is the work presented in [63]. It uses a modified Web browser to perform routing decisions at the client side. The browser downloads an applet which the service provider needs to implement to realize service specific routing. This approach creates increased network traffic due to applet transmission and potential control messages between the applet and the servers.

2.9 Conclusions

WebSeAl is a novel architecture for managing resources of Web sites consisting of a pool of geographically dispersed, replicated servers. Unlike most existing proposals, in WebSeAl it is the responsibility of the clients to route their requests to individual servers. This architecture supports geographic distribution, scales well with the number of users, and provides fault masking.

I proposed routing strategies for directing client requests to the most responsive servers. Unlike server side approaches, routing decisions are based not only on server load, but also on network traffic conditions. I also proposed strategies that can be used at the server side to induce efficient allocation of resources (load balancing) while clients make their routing decisions in a noncooperative manner. Motivated by recent studies on game-theoretic aspects of networking, I proposed a pricing mechanism that provides incentives to the clients to route their requests in a way that is deemed efficient by the service provider.

I have implemented a prototype system based on this architecture and its have validated its functionality through a series of experiments. These results indicate that WebSeAl can deliver significant performance gains while imposing minimal overhead.

Chapter 3

Metacomputing on the Web

3.1 Introduction

Over the last few years, the Internet has grown rapidly connecting millions of mostly idle machines. Its latest reincarnation as the Web has greatly increased its potential for utilization in diverse settings, including its potential to be used as a gigantic computing resource. On the other hand, utilization of local area networks as a parallel computing platform has been attractive for many years. There have been numerous research projects aimed at this goal. Based on their success, and as a natural evolutionary step, attempts have been made to extend existing systems from local area networks to wide area networks.

However, utilizing the Web as a metacomputing resource introduces new difficulties and problems different from those that exist in local area networks. First, many of the challenges that have been looked at individually (e.g., security, programmability, and scheduling) need to come together in a comprehensive manner. And secondly, the Web invalidates many of the assumptions used to build parallel environments for workstation clusters. For example, there is lack of a shared file system, no one has accounts on all connected machines, and it is not a homogeneous system. An environment to effectively utilize the Web as a metacomputing resource needs to address the following important issues: 1) programmability, 2) dynamic execution environment, 3) heterogeneity and portability, and 4) security.

Programming (Virtual Machine) Model: Generally, neither the programmer nor the end-user wants to deal with a dynamic and an unpredictable environment such as the Web. In fact, users are not interested in knowing whether their programs are running locally or remotely (perhaps in a distributed manner). For the Web to become an effective computing platform, the programming model needs to be decoupled from the dynamics of the execution environment. That is, programs need to be developed for a uniform and predictable virtual machine; the runtime system needs to realize the virtual machines. Otherwise, the task of program development becomes nearly intractable.

Dynamic Execution Environment: The Web is a dynamic environment comprised of many administrative domains; for example, machines become available and unavailable abruptly and network delays are unpredictable. Existing systems address these issues by providing some level of load balancing and fault masking. However, the extent of uncertainty is much greater on the Web.

Heterogeneity and Portability: The Web contains different types of hardware, running different operating systems, connected with different networks. Heterogeneity and portability are imperative to encompass the Web. Current heterogeneous systems are low level and generally employ a message-passing paradigm. High level systems based on virtual shared memory generally do not support heterogeneous environments.

Security and Accessibility: Cooperative work over the Web requires many facets of security measures. People need reassurance to allow "strangers" to execute computations on their machines. On local area networks this is accomplished by an administrator maintaining user access-rights and user accounts. Thus, the network becomes available only to a trusted set of users. This is not a feasible solution for the Web. In an ideal situation, any machine on the Web should be able to contribute to any ongoing computation on the Web.

A comprehensive solution for the Web requires that all of these issues be resolved. No system currently addresses all or even a majority of these. I present a system called *Charlotte* which addresses these issues.

The research leading to Charlotte started as theoretical work where provable methods for executing parallel computations on abstract asynchronous processors were developed [40, 38, 2]. The outline of the virtual machine interface to the actual system was proposed in [37]. Theoretical results were then interpreted in the context of networks of workstations in [20]. The above were significantly extended and validated in the *Calypso* [4] system which provides a virtual machine interface and a run-time system targeting homogeneous networks of workstations. This separation of programming and execution environment and the techniques employed in Calypso for load balancing and fault tolerance are also incorporated in Charlotte. We build on these and other complementary research efforts to offer a unified programming and execution environment for the Web. Our work on Charlotte has resulted in several original contributions summarized below:

- Charlotte is the first environment that allows *any machine on the Web to participate in any ongoing computation.* Two key factors make this possible. First, the system does not require a shared file system, nor does it require the program to reside on a local file system before a machine can participate in a computation. The Charlotte runtime system transmits the program to participating machines. Second, it is not necessary for a user to have an account (or any other type of privilege) to utilize a machine on the Web. The decision to involve a machine in a computation is made by the owner of that machine. These factors mean that potentially *any* machine can contribute to any running Charlotte computation on the Web.

- A novel technique for providing a *distributed shared memory (DSM) abstraction without relying on operating system or compiler support*. Current techniques for shared memory require either support from the operating system (in the form of setting page access-rights) or a compiler (to generate the necessary runtime code at each memory access). Shared memory systems based on operating system support are neither system independent nor safe. Compiler-based shared memory systems are tied to a particular programming language and a particular target language. Our approach does not suffer from these limitations, which makes it possible to provide virtual shared memory at the programming language level.

- Two integrated techniques, *eager scheduling* and *two-phase idempotent execution strategy* are used for load balancing and fault tolerance. We extend previous work originally developed in a different setting to deal with the dynamics of the Web.

- We leverage existing isolated contributions by providing a *unified and comprehensive solution*. Java is used for heterogeneity and portability. Charlotte's programming environment can be conceptually divided into a virtual machine model and a run-time system. The virtual machine model provides a reliable shared memory machine to the programmer and isolates the program from the execution environment. The run-time system realizes this model on a set of unpredictable, dynamically changing, and faulty machines.

- Charlotte is the *first programming environment for parallel computing using a secure language*. The Java programming language guards against mischievous code attacking local resources. Charlotte is built on top of Java without relying on any native code. This means that a Charlotte program provides the same level of security as a Java and Java-capable browsers.

The remainder of this Chapter is structured as follows: Section 3.2 describes Charlotte's virtual machine model and Section 3.3 the syntax and semantics of Charlotte programs. I then present two integrated techniques used in Charlotte, describe its runtime system, and show how machines are matched with parallel computations. In Section 3.7, I discuss the distributed shared memory implementation in Charlotte. Section 3.8 presents experiments, followed by related work and concluding remarks on Charlotte.

3.2 Virtual Machine

If a parallel program is to utilize the Web, its execution environment is not known at development time—the number of available machines, their location, their capabilities, and the network cannot be predicted ahead of time. In general, a program's execution environment will differ for each invocation. To deal with the dynamics

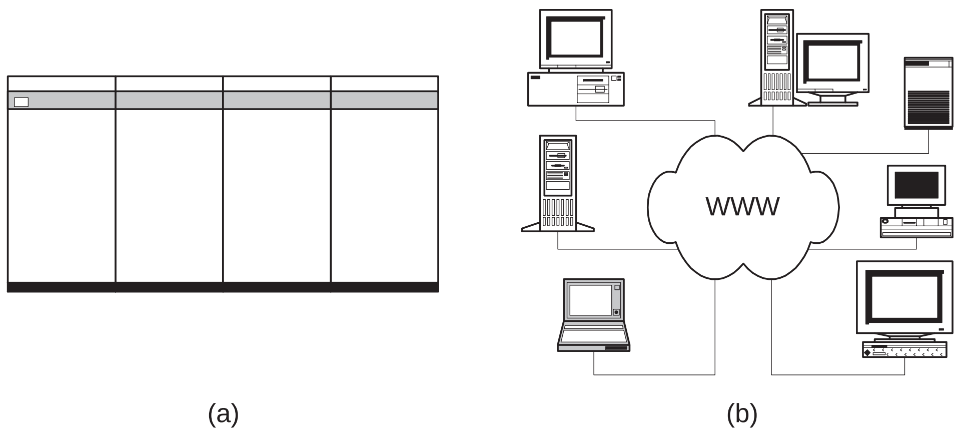


Figure 3.1: Perfect virtual machine vs. unreliable execution environment.

of the execution environment, either the programmer must explicitly write adaptive programs, or a software environment, such as a runtime system, must deal with the dynamics. We feel that the former solution puts too much strain on the programming effort. We conjecture that for an effective utilization of the Web, the programming model must be decoupled from the execution environment. Programs should be developed for a uniform and predictable virtual machine, thus simplifying the task of program development; the runtime system should implement virtual machines and deal with the dynamics of the execution environment.

Charlotte allows high level programming based on the parallelism of the problem and independent of the executing environment. Programs are written for a virtual parallel machine with infinitely many processors sharing a common namespace (see Figure 3.1(a)). The execution environment is the Web, an inherently unreliable distributed platform where different machines exhibit different performance, availability of machines change over time, machines fail unexpectedly, network delays vary significantly, and networks partition (see Figure 3.1(b)). The runtime system executes the given program on this platform; it performs load balancing among different machines, integrates newly available machines into a running computation, detects and removes failed machines from a computation, and maintains the coherence of distributed data transparently.

3.2.1 Programming Model

Charlotte programs are written by inserting *parallel* tasks into a *sequential* program. Computationally intensive work is performed by parallel tasks and high-level control flow, I/O, and other activities which are not computationally intensive are handled by the sequential code.

The execution of a parallel task is called a *parallel step* and the execution segment between two consecutive parallel steps a *sequential step*. The execution of a Charlotte program consists of alternating sequential and parallel steps. The execution always starts with a sequential step and ends with a sequential step, both or either one of which may be empty. The execution of a parallel step consists of any number of concurrent *routines*. A routine is analogous to a standard thread in Java, except for its capability to execute remotely. The number of routines is

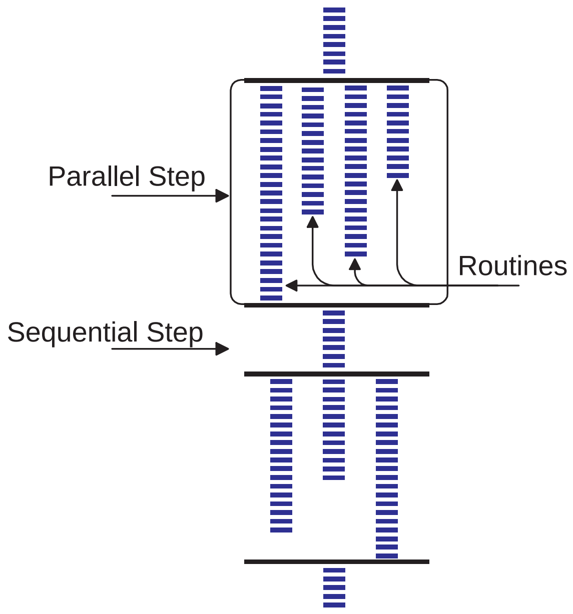


Figure 3.2: Charlotte's execution model.

independent of the actual number of machines available at runtime. Sequential steps consists of standard sequential Java code. Figure 3.2 illustrates a fragment of an execution.

3.2.2 Distributed Shared Memory Semantics

In a Charlotte program, the data is logically partitioned into *private* and *shared* segments. Private data refers to what is local to a routine and cannot be seen by others. The shared data is distributed and can be seen by others. For reasons to be discussed in Section 3.7, we have chosen to implement distributed shared memory within the language, at the data type level. That is, for every basic data type in

Java, there is a corresponding Charlotte data type implementing its distributed version. The consistency and coherence of the distributed data is maintained by the runtime system.

In an attempt to improve performance, many DSM systems have introduced multiple memory-consistency semantics [48]. The decision as to which consistency model is best suited for a particular application is left to the programmer. We feel this generally complicates a task the researchers were seeking to simplify. In Charlotte, we provide a single and intuitive memory semantics: *Concurrent Read, Concurrent Write Common* (CRCW-Common). This means that one or more routines can read a data variable, and one or more routines can write a data variable as long as they write the same value. More specifically, read operations return the value of the object at the time the parallel step began, and write operations become visible at the completion of the parallel step. Charlotte is not bound to this memory model, and, as a matter of fact, other memory semantics have been implemented in various versions of Charlotte, e.g. CR&EW. All the experiments in Section 3.8 were conducted using the memory semantics CRCW-Common.

The CRCW-Common semantics are implemented with the *Atomic Update* protocol. Intuitively, this means that write operations are performed locally and propagated at the completion of the routine. The advantages of this protocol are:

- Network traffic is reduced by packing many modifications into one network packet.

- Write operations are efficient since they are performed locally and do not require invalidation.
- Atomic execution of routines is guaranteed (since each routine is executed in an all-or-nothing fashion)
- Each routine can execute in isolation and can be verified independently of the execution order.
- The control logic in the coherence protocol is simple.

It should also be noted that the atomicity of updates is fundamental for the load balancing and fault masking techniques used in Charlotte, which will be discussed later.

3.3 Syntax and Semantics of Charlotte Programs

Charlotte does not modify the Java Virtual Machine, nor does it rely on a precompiler or require any kind of external runtime support. Its complete functionality is realized as a set of Java classes.

As mentioned in Section 3.2, Charlotte programs proceed with alternating sequential and parallel steps, and shared data can be accessed by concurrent routines.

Next, we will describe the syntax and semantics of sequential steps, parallel steps, and shared data types.

Sequential Steps: A sequential step of a Charlotte program consists of a sequence of standard Java statements.

Parallel Steps: Semi-formally, the syntax for a parallel step is as follows:

`parBegin();` *routine-list* `parEnd();`

The closing `parEnd()` serves as a synchronization barrier.

routine-list consists of sequence of `addDroutine()` statements. Any number of routines may be defined within one parallel step.

The syntax for `addDroutine()` is as follows:

```
addDroutine(Class class, int numRoutines);
```

The first argument to `addDroutine` is a subclass of `Droutine` (a Charlotte class) and is required to implement the function `drun()`. The second parameter defines the number of instances to run concurrently (which is passed to `drun()`, as described next).

`drun()` has the following syntax:

```
drun(int numRoutines, int id);
```

The first argument to `drun()` specifies the number of concurrent routines in the current parallel step (as passed to `addDroutine()`). The second argument is the routine identifier (in the range 0,...,`numRoutines`−1).

Shared Data Types: For every basic data type of Java, Charlotte implements a corresponding distributed version. For example, where Java provides `int` and `float` types, Charlotte provides `Dint` and `Dfloat` types. These are realized as standard Java classes. Objects of these classes can be instantiated in the standard way. For example, a two-dimensional array of floats of size $500 * 500$ could be instantiated as follows:

```
public Dfloat a[][] = new Dfloat[500][500];
```

Since Java prohibits operator overloading, distributed objects are accessed and modified with explicit function calls (*get()* and *set()*). For example, an element of the above declared array could be accessed and modified as follows:

```
float f = a[i][j].get()

a[i][j].set(1.23)
```

Charlotte's current implementation of distributed shared data requires a deterministic instantiation order. More specifically, all distributed objects must be instantiated in the same order at each site. The rationale behind this requirement will be discussed in Section 3.7. For now, it may suffice to remark that this can

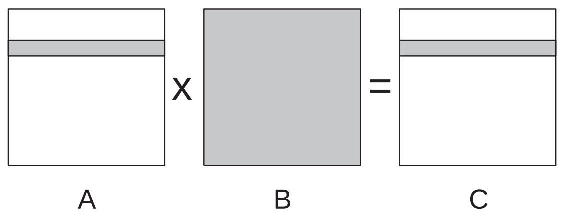


Figure 3.3: Matrix multiplication in Charlotte.

be easily achieved by instantiating all distributed objects in the constructor of a single class.

As an example of a Charlotte program, consider matrix multiplication. One possible approach to write a parallel matrix multiplication is to produce each row of the resultant matrix in a separate routine. To produce one such row, the routine needs the values of the corresponding row from one of the source matrices and all of the second matrix (see Figure 3.3).

The relevant fragments of the Charlotte program for matrix multiplication are shown in Figure 3.4. The important points to note here are:

- The program's parallelism is based on the parallelism of the problem at hand and not the execution environment. The same program can execute on one or any number of machines, depending on availability.

- The programmer does not need to be aware of the fact that this application can be executed in a distributed fashion, utilizing multiple machines on the Web.

```
import charlotte.*;
// ... code cut here ...

public class MatrixMult extends Droutine {
  public static int  Size = 500;
  public Dfloat a[][] = new Dfloat[Size][Size];
  public Dfloat b[][] = new Dfloat[Size][Size];
  public Dfloat c[][] = new Dfloat[Size][Size];

  public MatrixMult() {
    // ... code cut here ...
  }

  public void drun(int numTasks, int id) {
    float sum;
    for(int i=0; i<Size; i++) {
      sum = 0;
      for(int j=0; j<Size; j++)
        sum += a[id][j].get() * b[j][i].get();
      c[id][i].set(sum);
    }
  }

  public void run() {
    // ... code cut here ...
    parBegin();
    addDroutine(this, Size);
    parEnd();
    // ... code cut here ...
  }

  // ... code cut here ...
}
```

Figure 3.4: Charlotte matrix multiplication code: MatrixMult.java.

- Integrating machines into the computation, load-balancing, fault-masking, and data coherence are transparent.
- The services provided by Charlotte do not require any language or compiler modifications.

3.4 Eager Scheduling and TIES

The paradigm that Charlotte embodies was first described in [40]. That paper described a methodology for instrumenting general parallel programs to automatically obtain their fault-tolerant counterparts that could run on an abstract shared memory multiprocessing machine. While the solutions in [40] were formulated in the context of synchronous faults, they were later applied in [39, 38] to a certain variant of asynchronous behavior, i.e., could run on a machine whose processors take arbitrary amounts of time to execute each step. This research formed the basis of the ideas first incorporated into Calypso [4] and later in Charlotte.

A unified set of mechanisms, *eager scheduling* and *collating differential memory*, is used to provide the functionality of Charlotte. The *idempotence* property is fundamental in Charlotte: a code segment can be executed multiple times (with possibly some partial executions), with exactly once semantics.

The importance of idempotence, and the utilization of eager scheduling to take advantage of it, was discovered in [40] in an abstract context. The term "eager scheduling" was termed later. Eager scheduling is a mechanism for assigning concurrently executable tasks to the available machines. Any machine can execute any "enabled" task, independent of whether this task is already under execution by another machine.

The mechanism of collating differential memory provides logical coherence and synchronization while avoiding false sharing. It is an adaption and refinement of the *two-phase idempotent execution strategy* [40]. Memory updates are collated to assure exactly-once logical execution. This prevents false sharing and supports efficient implementation of idempotence in addition to other performance benefits which will be described later. Details about these mechanisms will be discussed in Section 3.7.

Charlotte relies on these three mechanisms to ensure that the available machines are *used where they are needed most and without delays associated with false sharing.* A "participating" machine is used to mask faults and/or increase the parallelism depending on a transient state of the system. A critical feature of Charlotte is that it is neither a fault-tolerant system extended for parallel processing nor a parallel processing system extended for fault tolerance. *A single unified set of mechanisms provides both parallel processing and fault tolerance.*

In summary, eager scheduling combined with TIES has the following properties:

- Any of the machines that are executing concurrent routines can fail or slow down at any time.

- As long as at least one "participating" machine does not continuously fail, all jobs will be completed.
- A crash-failed machine or a slow machine is transparently bypassed by faster ones, leading to a balanced system and fault tolerance.
- New machines can be transparently and productively integrated into a computation anytime, even in the middle of a parallel step.
- Computations do not stall while dealing with system's asynchrony and faults.

3.5 Runtime System

Charlotte provides three major services: (1) scheduling service, (2) memory service, and (3) computing service. In the current prototype, the scheduling service, the memory service, and parts of the computing service executing the sequential steps are fused into a single process—the *management service*, or the *manager* for short. The computing service performs the computationally intensive parallel routines; I refer to these processes as *workers*. A single manager and one or more workers together implement the virtual machine the program was written for.

Charlotte's runtime environment is the Web. In general, the execution of a Charlotte program is distributed over a dynamically changing set of machines. We

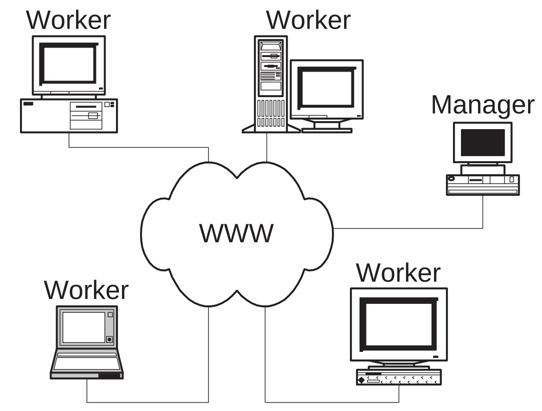


Figure 3.5: Charlotte's runtime environment.

assume that a user has one machine under her control and that this machine is highly reliable. The user, however, may wish to utilize other machines on the Web which are unreliable. The user does not need an account or any kind of privileges on these machines. Also, there is no need for a shared file system.

Charlotte employs a *volunteer-based* approach [54] to parallel computing on the Web. Specifically, a user can initiate a parallel computation on her machine, but the decision as to which other machines contribute is made by the owners of these machines. If a volunteer wishes to donate her resources to a parallel computation, she joins in and continues to contribute until either the computation ends or the volunteer decides to discontinue, which may happen at any given time. The manager executes on the user machine where the parallel computation was initiated, and the worker execute on volunteer machines. This setting is depicted in Figure 3.5. I will now sketch the overall execution strategy.

3.5.1 An Overview of the Execution

A Charlotte program is executed by exactly one manager process running on the user's reliable machine, and a dynamically changing set of worker processes running on arbitrary machines connected to the Web which may come and go, or slow down and speed up unpredictably, depending on the transient availability of resources and *not* on the properties of the computation.

During a computation, the manager executes the sequential steps. Parallel steps are executed by the available workers and "managed" by the manager. Each particular parallel step has some number of concurrently executable routines as specified by the `addDroutine` calls of the corresponding program. I will refer to each instance of a routine as a *job*. Each of the available workers contacts the manager and obtains a work assignment: an unfinished job. It then proceeds to execute it. Once finished, the worker reports the results back to the manager and requests another assignment. The manager keeps track of jobs (which ones have been assigned, which ones have been completed, etc.) and has the responsibility to assign more work. At first, it prefers to assign jobs which have not yet been assigned to any worker. But what should it do when the number of available workers exceeds the number of jobs? Or when all jobs have been assigned but not

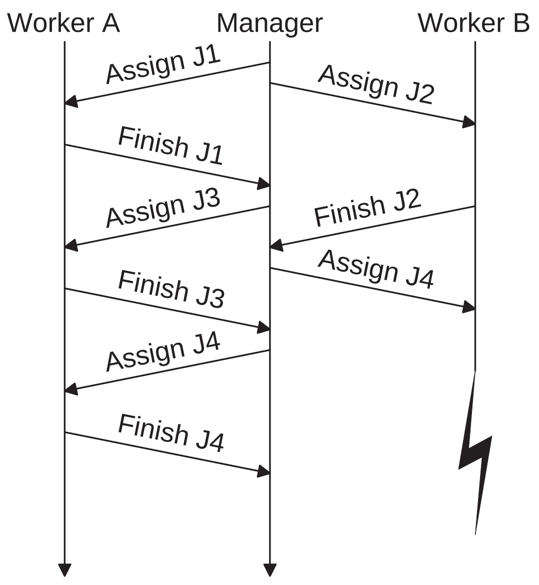


Figure 3.6: Eager scheduling example.

finished, and there are idle workers eager to work? Eager scheduling implies that if there is unfinished work, and there is an idle worker, then the work will be assigned to that worker. The manager simply assigns work using eager scheduling until all jobs are completed, at which point that parallel step is considered complete.

Although I have not discussed what it means for a worker to do useful work, it should be clear how eager scheduling leads to both, load balancing and fault tolerance in a unified solution. A crash-failed worker is a special case of a slow worker—it is an infinitely slow worker—and a fast worker will never wait for a slower one when it can overtake it. The "unnecessary" computation is generally small, and in practice it is essentially free.[1] In experimenting with various pro-

[1]Note that the replicated work is assigned to workers that have "nothing else to do" beyond partici-

grams, we were *fully* charged for this extra work, and we observed that the overhead was, in fact, very low. In Section 3.8, I will describe several experiments and report the results Charlotte achieved.

Figure 3.6 illustrates eager scheduling through an example. Two machines, *Worker A* and *Worker B*, work on a parallel step consisting of four jobs of the same length. Worker A is fast and completes job J1 and starts working on J3, before the slower Worker B finished job J2. However, while executing job J4, Worker B crashes. Once Worker A finishes J3, the manager eagerly schedules J4 to Worker A. This illustrates that Charlotte does not require any failure detection of worker machines; failures are automatically handled by eager scheduling.

For a worker to do useful work, it must not only perform the computation that is specified by its assignment, but it must perform operations on the manager's data-set, i.e., share data, and return the results back to the manager. Shared data is demand-paged in a manner analogous to that of virtual memory systems. But, as stated before, this alone does not ensure idempotence of updates and memory coherence in presence of eager scheduling. The additional techniques required for this are discussed next.

pating in the current parallel step.

3.5.2 A Sample Execution

As described above, a Charlotte program is executed by a single manager and a dynamically varying set of workers.

When the computation starts, the manager executes the sequential code until it reaches the first parallel step. It then suspends its execution and gets ready to manage the execution. As an example, consider the following program fragment:

```
...
parBegin();
addDroutine(m, n);
parEnd();
...
```

Assume that *m* is an instance of a class called *MyClass* and the variable *n* is evaluated to 4. The manager then prepares a table called the *progress table* and initializes it as follows:

Step	Class	Width	Identification	Started	Finished
2	MyClass	4	0	NO	NO
2	MyClass	4	1	NO	NO
2	MyClass	4	2	NO	NO
2	MyClass	4	3	NO	NO

The third row of the table, for instance, indicates that this is step number 2; that a job defined by the `drun()` function of the class `MyClass` needs to be computed; that there are four such jobs; that this rows describes the third out of four sibling jobs (numbered 0, 1, 2, 3); that no worker has started working on this job; and that its computation has not yet finished.

Assume that the first three jobs have been assigned, and that the first and the third job are still being worked on, but the second has already finished. This is reflected by the following table:

Step	Class	Width	Identification	Started	Finished
2	MyClass	4	0	YES	NO
2	MyClass	4	1	YES	YES
2	MyClass	4	2	YES	NO
2	MyClass	4	3	NO	NO

Charlotte utilizes a simple version of *eager scheduling* as follows. The manager listens to workers requesting assignments. It assigns to each free worker a job that has not yet been finished—among all such jobs it assigns one that has been assigned the least number of times.

I now turn to the description of a worker. The worker, executing within a browser, contacts the manager for work and initializes the shared data as "not valid". I will describe the association of workers and manager in Section 3.6 and the issues regarding shared data in Section 3.7. The manager sends the worker an assignment specified by the 3 parameters `class`, `width`, and `identification`. The worker now starts executing this assignment: it performs a call to the `drun()` function of `class` with the parameters `width` and `identifier`. During the execution, the first time a worker accesses a "not valid" shared variable, an internal signal is raised. Charlotte's signal handler fetches the appropriate data from the manager, installs it in the worker's name space, and marks it as "valid". Then. the computation proceeds. When the worker finishes the job, it identifies all of its dirty data and sends them to the manager. It then re-invalidates its shared data and contacts the manager for another assignment.

The manager accepts the first completed execution for each job and discards subsequent ones. Late updates from workers are easily recognized and ignored (e.g., an update for a job assigned in step 2 but arriving in step 4).

The manager buffers the updates until the end of a parallel step, at which time all updates are performed. Different parts of the shared data can be updated by different workers, as long as the CRCW-Common condition is met. Note that this condition is an aspect of the logical program design, and is independent of the various workers' assignments. In a parallel step, values read by a worker are those existing at the beginning of the step, and the updated values are readable only at the beginning of the next step.

The shared memory is always logically coherent as far as the program is concerned yet there is no need for expensive mechanisms such as *distributed locking* or *page shuttling*. This is due to the exploitation of the synchronization primitive which is a side effect of the language construct, as well as transmitting changes only. Also, the collating technique (buffering updates, accepting the first update, discarding the others) in fact implements *two-phase idempotent execution strategy.* As a consequence, correctness is assured in spite of the multiplicity of executions.

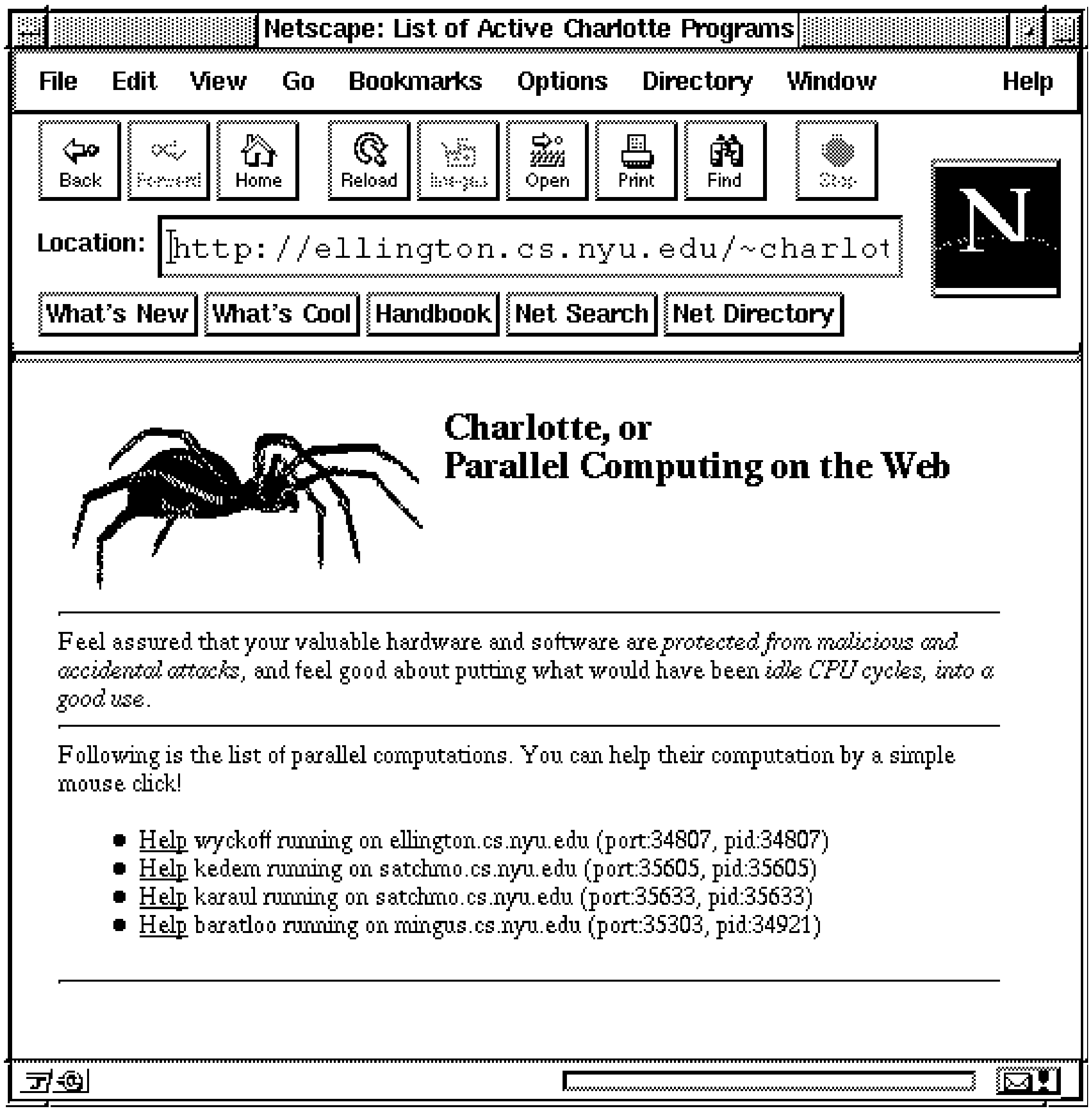
We have added the following technique to Charlotte to improve its performance. Memory that has been paged-in by a worker is kept valid as long as possible. For instance, the manager knows in which step an object had been modified last. For instance, if some object was modified last in step 4, it was read by some worker in step 6, and that worker is working on a job in step 8, then the worker does not fetch the object but accesses its cached copy. This is a low cost (almost free) strategy which results in a significant performance benefit. Note that read-only shared objects are fetched by a worker at most once, while write-only shared objects are never fetched. Modified shared objects are re-fetched only when necessary. To achieve this, invalidation requests are piggybacked on the work assignment messages, bearing only little additional cost. The programmer *does not* declare the type of coherence or caching technique to use, rather, the system adapts dynamically.

It should be clear from the above description that workers (and their location) and jobs are not related in any fundamental way. In fact, the syntax of Charlotte *does not even allow* the programmer to specify the workers or how to distribute the data. However, Charlotte has recently been extended with "colocation" techniques using annotations [35] to allow the manager to assign a job based on its expected data access patterns and the cached shared data of candidate workers.

3.6 Matching Workers with Computations

It is very likely that at any moment there are idle machines on the Web willing to help in some computation, and that there are computations that could utilize these machines. Both these sets are dynamic. Now, the difficulty lies in associating an idle machine with a computation. It is only when this association has been established, and the necessary program fragment has been made accessible, that an idle machine can contribute to the computation. However, the lack of trust in allowing "strange programs" to execute on local hardware is a serious issue. The complete problem is of great importance since it lies at the core of providing an effective metacomputing environment.

As a first version, we have adopted a solution which does not scale for settings such as the World Wide Web, but it is an effective solution for our network at New York University. Oversimplifying it, when a Charlotte program reaches a


Netscape: List of Active Charlotte Programs
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
Location: http://ellington.cs.nyu.edu/~charlot
What's New What's Cool Handbook Net Search Net Directory

**Charlotte, or
Parallel Computing on the Web**

Feel assured that your valuable hardware and software are *protected from malicious and accidental attacks*, and feel good about putting what would have been *idle CPU cycles, into a good use*.

Following is the list of parallel computations. You can help their computation by a simple mouse click!

- Help wyckoff running on ellington.cs.nyu.edu (port:34807, pid:34807)
- Help kedem running on satchmo.cs.nyu.edu (port:35605, pid:35605)
- Help karaul running on satchmo.cs.nyu.edu (port:35633, pid:35633)
- Help baratloo running on mingus.cs.nyu.edu (port:35303, pid:34921)

Figure 3.7: Sample list of active Charlotte programs.

parallel step, it registers itself with a specific daemon process. This action creates an entry in a URL homepage (see Figure 3.7). Any user on our network can visit this homepage using any Java-capable browser and see the list of active programs. If the user wishes to donate some of the CPU power of her machine, she can simply click on an entry. This will load the required code to the user's machine and start assisting the ongoing computation.

To overcome the limitations of this solution, we have recently incorporated a directory service into Charlotte to match users with computations. It provides a scalable solution and works fully within browsers. I will discuss the details of this directory service in Section 4.4.

Since Charlotte is entirely implemented in Java, it provides the same security guarantees as Java. Java guarantees the protection of local resources from programs. Although there seem to be security holes in the current implementation (e.g., [21]), these are constantly being addressed. Charlotte will transparently take advantage of these improvements. Once users stop being afraid of programs that come over the network, we feel that they will have more incentive to allow others to use their idle CPU cycles

It is important to stress that Charlotte supports heterogeneous systems, which makes it possible for any idle machine to execute a parallel computation alongside any other—a necessity for the Web.

3.7 Distributed Shared Memory

To provide the abstraction of a single address space to multiple programs running on different machines, distributed shared memory systems must detect accesses to shared data and propagate updates. For a shared memory system to be realized over the Web, hardware and operating system independence is a necessity, and language/compiler independence is desirable. In Charlotte, we achieve both.

3.7.1 Existing Techniques

There have been two approaches to implement distributed shared memory at the software level: one relies on virtual memory page protection and the other on a compiler to provide software write detection.

Traditional software-based shared memory systems rely on virtual memory page protection [47, 4, 41]. Detection of a data access is done by protecting the virtual memory pages and catching the page-fault signal generated by the operating system. A write operation sets a dirty-bit for each page, indicating that the change needs to be propagated. The granularity of shared data segments is determined by the system—a virtual memory page. This can result in *false sharing* when the same memory page is accessed independently by multiple processes. Since page size and data format vary across different machines and operating systems, this is not a viable option for a heterogeneous environment such as the Web. Furthermore, Java does not support the required operating system calls to protect a virtual memory page and to catch a page-fault signal.

Another technique to implement DSM relies on compiler and runtime support [64]. A compiler inserts the necessary code to detect and service an access to the shared memory region. This approach has the advantage that the granularity can be controlled, meaning that it alleviates false sharing. The disadvantage is that it requires the system to continuously evolve with the language upon which it is based. We feel this is not desirable.

Charlotte's shared memory abstraction is neither operating system nor compiler based. We were introduced to the feasibility of software write detection by the work in [64]. However, rather than using a compiler, we chose to realize the shared memory abstraction through a shared name-space. This method combines the advantages of both methods mentioned above:

- Programs can run on any combination of hardware and operating systems where Java is available.
- Charlotte does not need to evolve with every change to the Java language and compilers.
- The shared memory abstraction is available in heterogeneous environments.
- The granularity of shared data can be controlled dynamically.

3.7.2 Implementation

Charlotte's distributed shared memory is implemented within the language, at the data type level; that is, through Java classes. In addition to a `value` field, each shared data object maintains its state which can be one of `not_valid`, `readable`, or `dirty`. A `not_valid` state indicates that the object does not contain a meaningful value; `readable` indicates that the object contains a meaningful value which can be used in a read operation; and `dirty` indicates that the local value is meaningful and that it has been modified.

Distributed objects are read and written through class member functions. A read operation on a `readable` or a `dirty` object just returns its `value`, which is locally available. Otherwise, its value is retrieved from a manager process over the network and its state set to `readable`—this corresponds to demand-paging in traditional virtual memory page based systems. On a write operation, the `value` is modified and the object state set to `dirty`—this corresponds to setting the dirty bit in traditional systems. A `dirty` object propagates its value at the end of the job to the manager.

A problem in this implementation occurs when an instance of an object in `not_valid` state is accessed. How does *this particular instance* convey its identity to the manager to retrieve its value? After all, the manager has many instances of the same class. Our solution is based on a unique identifier, assigned deterministically,

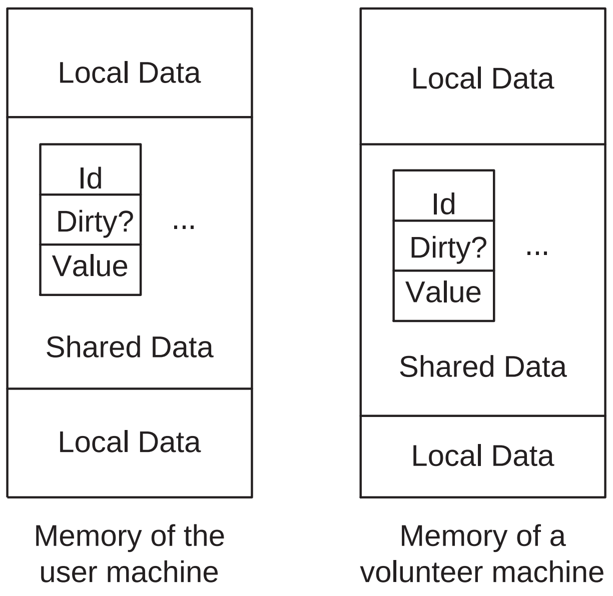


Figure 3.8: Distributed objects in Charlotte.

based on instantiation order. This requires that distributed objects be instantiated in the same order at each site. For simplicity, we choose to instantiate all the distributed objects in the constructor of a single class. This guarantees that each replica of a distributed object is assigned the same identifier at every site. This scenario is depicted in Figure 3.8.

We have implemented two techniques to improve the performance of shared memory in Charlotte. First, a read operation does not result in transferring the value of a single item. Instead, multiple items are shipped in a single network packet. The size of each packet can be set dynamically at runtime, thus adjusting the granularity of shared memory. Second, the information as to which objects are invalidated is piggy-backed with the manager's job assignment. Thus, each worker page-faults on read-only data items at most once.

The issue of choosing the size of the packet, though, can be difficult. Larger packet sizes reduce data request frequency, but increases the potential for false sharing. Similarly, smaller packet sizes reduce the chance for false sharing, but increase the data request frequency. Ideally, the whole "read set" should be sent to a worker along with the job assignment. This could be achieved by using *annotations*, for example, allowing the programmer to give the runtime system hints about data access patterns. This would serve two purposes: (1) the latency to wait for data requests is avoided, and (2) false sharing is avoided. Furthermore, if it is guaranteed that the specified read set is complete, the overhead of catching page-faults can be eliminated altogether. These changes, along with various intermediate solutions, have recently been incorporated into Charlotte. The details can be found in [35].

In summary, our approach for realizing distributed shared memory on loosely coupled machines does not rely on virtual memory pages (operating system) nor a compiler. Similar to compiler based systems, it runs in user-space, supports variable data granularity, and avoids false sharing. At the same time, it is not dependent on any particular implementation of the language or compiler. It is important to note that this technique is not tied to any particular memory coherence model—it is a general technique which can be used to implement many different memory coherence models.

3.8 Experiments

Here I present performance results. To evaluate the performance of Charlotte, we chose a scientific application from statistical physics—computing the 3D Ising model [10]. This is a simplified model of magnets on a three dimensional lattice which can be used to describe qualitatively how small systems behave. Computing the Ising model involves an exponential number of independent tasks and very little data movement.

All experiments were conducted using interpreted Java. A C program runs an order of magnitude faster than an interpreted Java program. However, Java compilers are becoming more and more available and they provide performance much closer to C. We expect these results to carry over transparently.

3.8.1 Setup

We wrote a single Charlotte program and ran it on sets of workstations under different patterns of slowdowns, failures, and recoveries. The same program was used in each case and the runtime parameters did not change. Results are shown for the following cases, all for the same Charlotte program:

1. The program running on a set of identically behaving, otherwise unused workstations. We wanted to measure speedups when there were no failures, no slow-downs, and no need for load balancing or fault tolerance (of course, the runtime system "did not know this.")

2. The program running on various combinations of fast and slow machines.

3. The program running on a set of machines, some of which crash-fail during the computation.

4. The program running on a set of machines, and additional machines becoming available during the computation.

All experiments were conducted as follows:

- The identical Charlotte program was used for *all* tests.

- The machines used were up to ten identical Sun SparcStation 5 workstations, connected by a 10Mbit Ethernet. The network was disconnected from the rest of our network to eliminate any outside traffic.

- All experiments computed the Ising model with a period of 23.

- The program consisted of a sequential step performing certain initializations and starting a timer, followed by a parallel step with 23 routines, and a

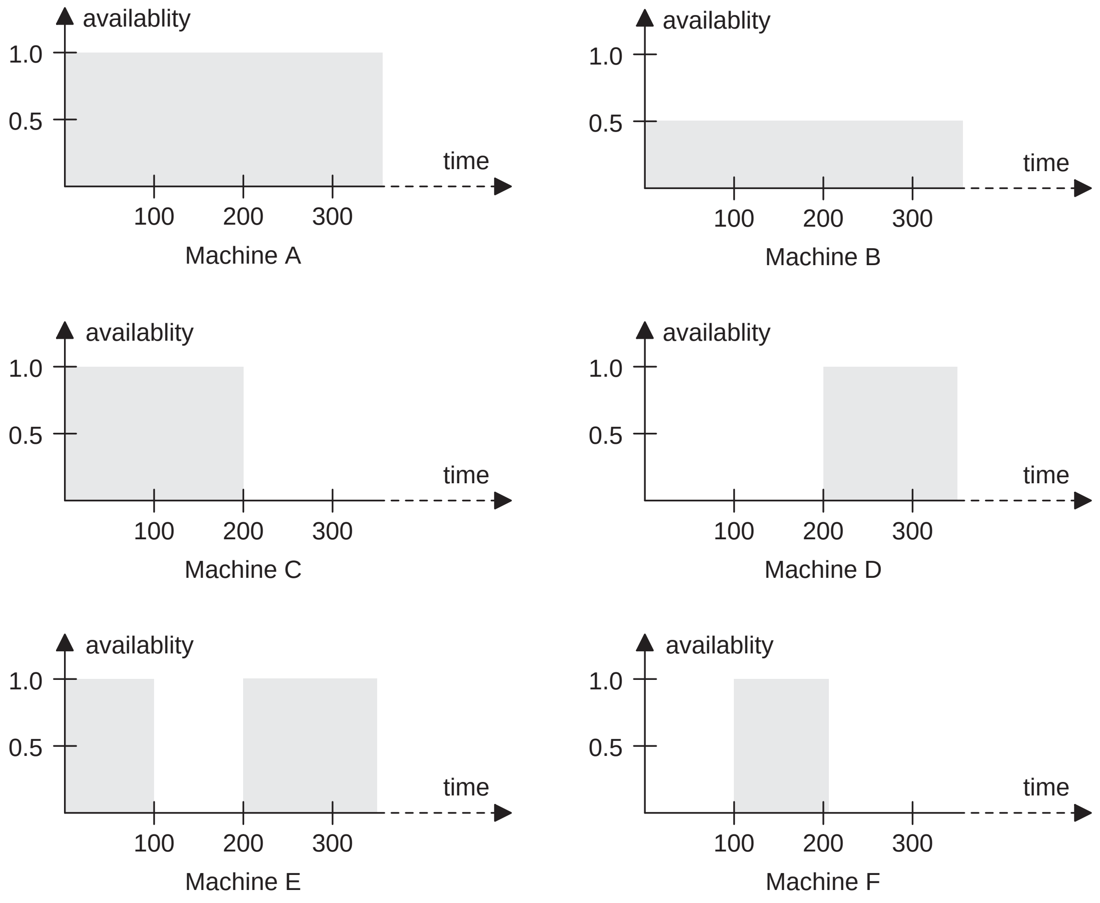


Figure 3.9: Profiles of machines available for the experiments.

closing sequential step which was empty except for stopping the timer and determining the overall time the parallel step took.

To conduct the experiment, I defined six machine *profiles*. Each machine profile determines its behavior. Here are the descriptions of each machine profile, see also Figure 3.9.

- *Machine A* is available to the computation 100% for the duration of the computation. This is a machine which does not fail or slow down during the execution. I call this a *fast machine*.

- *Machine B* is available to the computation 50% for the duration of the computation. This is a machine which contributes to the computations at half its regular speed. This is achieved by running a background process (called `hog`), at high priority, which runs for a second and sleeps for a second. I call this a *slow machine*.

- *Machine C* crash-fails after 200 seconds from the start of the computation. This is achieved by killing the worker process manually at the corresponding time.

- *Machine D* joins the computation after 200 seconds from the start of the computation. This is achieved by starting the worker process manually at the corresponding time.

- *Machines E* crash-fails after 100 seconds and joins the computation after 200 seconds from the start of the computation. This is achieved by killing and starting the worker process manually at the corresponding times.

- *Machines F* joins the computation after 100 seconds and crash-fails after 200 seconds from the start of the computation. This is achieved by starting and killing the worker process manually at the corresponding times.

All times reported are "wall clock" or elapsed times, *not* CPU or virtual times. Since we cannot improve the performance of sequential tasks, we measured the times from the start of the parallel step until the end of the parallel step. The manager runs on a machine which does not fail, but may slow down. A single worker runs on each participating machine, including the machine the manager is running on. It should be stressed that at time $t = 0$, the workers have no shared data and at the end of the computation, $t = T$, the manager has received and processed all the outputs from the workers. Thus, we will account for all the costs associated with the execution of the parallel steps.

In each experiment, we use up to ten machines from the six profiles. Whenever a machine is "available" to us, we are charged for its use, regardless of whether we are in fact able to benefit from it or not.

In addition, all overhead (networking, file access, swapping, updating memory, etc.) are included in the charges, regardless of whether we have control over it or not. Hence, the results are based on quite conservative assumptions.

3.8.2 Results

We conducted three series of experiments and I will describe them in turn. First, we study the performance and overhead of Charlotte. Second, we examine the

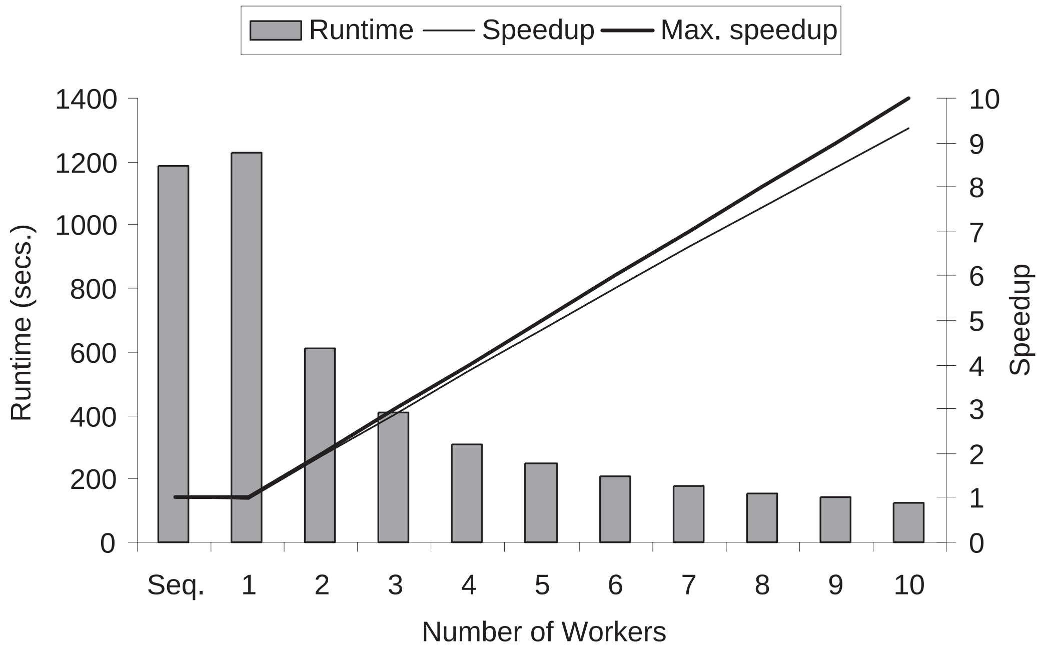


Figure 3.10: Scalability experiment of a Charlotte Ising model program.

utilization of slow machines in a computation. In particular, we are interested to see whether the addition of slow machines will affect the overall performance. Third,we analyze how well the system can integrate machines into an ongoing computation, and how efficiently failures can be masked. The results are graphed showing the total time, the achieved speedup, and the number of equivalent perfect machines (which represents an upper-bound for the speedup).

1. This series of experiment shows the overhead of Charlotte. A sequential Java program to compute the Ising model ran in 1,186 seconds. The equivalent

Charlotte program ran in 1,230 seconds on one machine of profile A (the manager process and one worker process ran on the same machine). This corresponds to 96% efficiency. The same program on two machines of profile A (one machine running the manager and one worker, and the second machine running the second worker) ran in 609 seconds. This represents a speedup of 1.95 and 97% efficiency compared with the sequential Java program. Figure 3.10 shows the performance for 1 through 10 workers. Given the high level programming model and a need for optimization, we were satisfied that the program achieved 93% efficiency with 10 machines. This is competitive with other systems that do not provide load balancing and fault masking like Charlotte.

2. In the second set of experiments, we evaluate how efficiently Charlotte can handle an environment composed of some fast and some slow machines. This models a "real" setting on the Web. We used machines of profile A and profile B. We performed six tests. In all tests, the number of actual machines varied from five to ten, although the effective number of fully available machines was always five. For example, we ran a test with three machines that were available 100% of the time (profile A) and four machines that were available 50% of the time (profile B). As Figure 3.11 indicates, Charlotte's load balancing technique was effective in this environment.

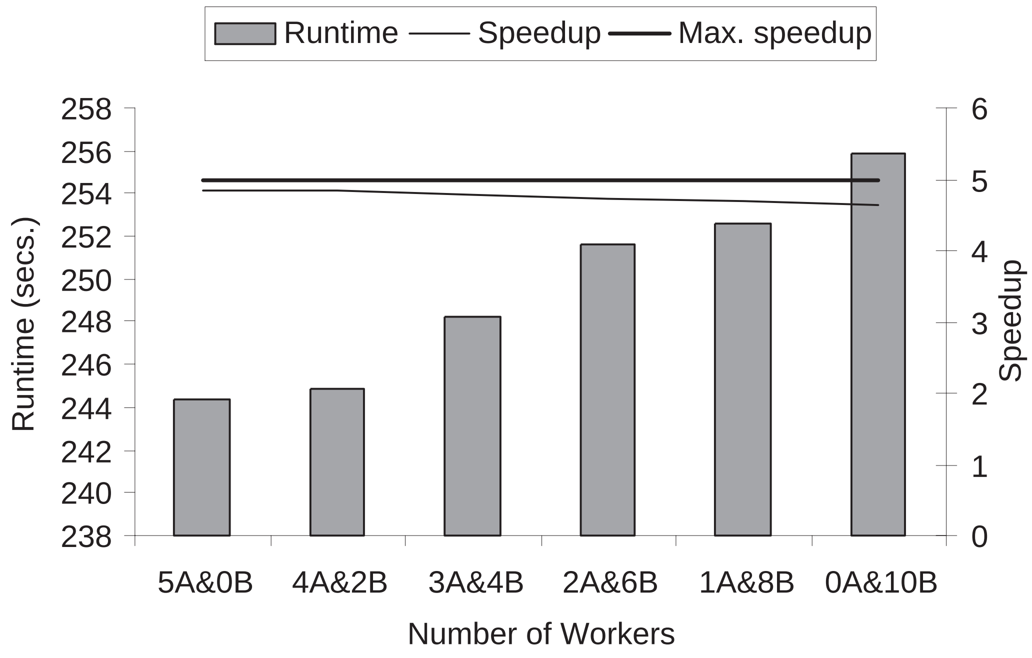


Figure 3.11: Load balancing experiment of a Charlotte Ising model program.

3. In the final experiment, we measure how well Charlotte handles failures and how effectively a machine can be integrated into a running computation. In this test, we ran the program on seven machines—three machines of profile A (one of these running the manager and a worker), and one each of profiles C, D, E, and F. This corresponds to the following scenario: started with five machines; after 100 seconds, one worker was crashed and instantly a new worker started; after another 100 seconds, two workers were crashed and instantly two new workers started. At any time during the computation the effective number of fully available machines was five. This program completed in 275 seconds, as opposed to 248 seconds in a perfect setting, i.e. using five machines of profile F. This indicates that Charlotte performs well in a dynamic situation.

Other applications, such as matrix multiplication and Mandelbrot, have been parallelized using Charlotte. The work described in [35] contains further performance results for Charlotte. It also includes an analysis of the overhead Charlotte imposes, and proposes certain optimizations.

3.9 Related Work

PVM [57] and MPI [28] are representatives of message passing systems. They provide portability and good performance, but they are low level. Systems such as CARMI [53] and [33] augment PVM's functionality with resource management services. However, they are limited to local area networks: there is no support for dynamic load-balancing and fault masking, they require the program to reside at each site (or a shared file system), and the user or the system needs an account on each machine participating in the computation. These factors severely limit their use as a metacomputing framework for the Web.

Another class of systems for distributed computing focuses on providing distributed shared memory across loosely-coupled machines. IVY [47] and TreadMarks [41] are representatives of such systems. Cilk [11] is a comprehensive system providing resource management and fault-tolerance in addition to DSM. However, it makes similar assumptions about the file system and user privileges as message passing systems, which limits its applicability to the Web. In addition, DSM systems, in general, do not work on heterogeneous environments.

Recently, with the introduction of Java, another class of systems is becoming available. This class includes: JPVM [24], Java-MPI [59], ATLAS [3], JavaParty [52], and ParaWeb [12]. Similarly, there have been proposals for Web-enabled virtual machines as a basis for a High Performance Computing and Communications (HPCC) platform [27], to use Java in a SPMD programming model [31], and to extend Java with global pointers and remote service request mechanisms from the Nexus communication library [26].

Both JPVM and Java-MPI provide a message passing interface to Java stand-alone applications, but not applets. ATLAS programs are distributed and load-balanced using Cilk's work-stealing scheduling. It requires daemon processes as compute servers and relies on native code, destroying the secure program execution guarantees. JavaParty mainly targets clusters of workstations. It enhances Java with remote objects and uses a preprocessor to generate data distribution and data migration code. JavaParty requires a running Java process as a runtime environment, creating an administrative burden. ParaWeb allows threads to run remotely and gives the abstraction of a single shared memory. Essentially, it is a parallel implementation of the Java Virtual Machine, but requires modifications.

Unfortunately, all these projects fail to take advantage of Web browsers' ability to download and execute applets in a secure fashion. This is a crucial requirement if a system is to entice users to donate their resources. Recently, several projects have appeared which provide a parallel computing environment for the Web without sacrificing the necessary security guarantees. Javelin [14] provides brokering functionalities for computational resources and a layer supporting the implementation of parallel models in Java. Bayanihan [54] proposes a volunteer-based approach and uses a technique similar to eager scheduling to assign jobs to volunteers. Ninflet [58] supports RPC-based computing allowing clients to invoke remote methods from various languages such as Fortran, C, and Java; it provides its own security manager to address security concerns.

3.10 Conclusion

Charlotte supports some of the key functionality critical for harnessing the Web as a metacomputing resource for parallel computations. It provides programmers with a convenient and stable virtual machine interface to the heterogeneous and unpredictable execution environment. The programming model is based on shared memory and employs the emerging Java standard enhanced with a few classes for expressing parallelism. Thus, heterogeneity and security are provided to the extent supported in Java. Furthermore, the runtime environment realizes automatic load balancing and fault tolerance, both critical to the effective utilization of the Web.

Chapter 4

Network Computing with Java Applets

4.1 Introduction

The Web has the potential of integrating remote and heterogeneous computers into a single global computing resource for parallel and collaborative work. While parallel computing on workstation clusters is common practice with systems such as PVM [57] and MPI [28], this is not the case for computing over the Web. Similarly, while collaborative work on Intranets is supported by systems like CORBA [56] and DCOM [49], this does not hold true for the Web. Some of the obstacles common to both Web-based parallel computing and collaborative work are the heterogeneity of the participating systems, difficulties in administering distributed applications, security concerns of users, and matching of applications and users.

The Java programming language in combination with Java-capable browsers have successfully addressed some of these problems. Java's platform independence solves the problem of heterogeneity. The growing number of Java capable browsers able to seamlessly load applets from remote sites reduces administration difficulties. The applet security model, which in most parts enables browsers to execute untrusted applets in a trusted environment, alleviates some of the users' security concerns. It is because of these reasons that Java applets are a good candidate for building Web-based systems, and browsers the perfect candidate to seamlessly bring distributed computing to every-day users.

In this Chapter, we focus on applications composed of Java applets and designed to run within Web browsers. As a result of limitations put on applets, the architectures of many existing systems have the following characteristics: (1) a stand-alone Java application running on the same host as an HTTP server; and (2) users anywhere on the Internet joining a stand-alone application by loading an applet using an a priori known URL and executing it within a browser. In an ideal situation, (1) the stand-alone Java application should not be tied to an external HTTP server and should be allowed to execute anywhere on a network; and (2) users should not be required to have a priori knowledge of the URL and should be able to locate these applications in a seamless fashion via browsers. Knitting-

Factory is an infrastructure for building such systems. It provides two integrated services:

- *KF Directory Service*, a distributed name service to assist users in finding networked applications on unknown hosts. *KF Directory Service* makes it possible for the user to find such applications on the Web by performing a lookup at any site offering a *KF Directory Service*. Since we are mostly concerned with applications which register and deregister frequently, the challenge is to provide this functionality in a decentralized fashion, but more importantly, such that it works with Web browsers.
- *KF Class Server*, an embedded class server to eliminate the need of external HTTP servers. A user who may wish to initiate a distributed application may not have access to a host running an HTTP server and it may be inconvenient or impossible to install such a server on the local host. *KF Class Server* provides a light-weight solution to allow users to initiate a distributed effort on any host.

The rest of this Chapter is organized as follows. Next, I give an overview of related projects to highlight how KnittingFactory can assist in overcoming some difficulties associated with applets. Section 4.3 describes the philosophy and design of KnittingFactory. Sections 4.4 and 4.5 describe the implementation and sample applications of the respective services. A more detailed description of the implementation can be found in [5]. I will close with concluding remarks on this project in Section 4.6.

4.2 Related Work

There are various areas which aim to take advantage of Java applets. In this Section, I will focus on two areas, *parallel computing* and *collaborative applications* on the Web. I will present a few projects in these areas and point out the commonalities of these projects. I will then extract common shortcomings and argue that there is a need for infrastructure support to overcome these shortcomings.

4.2.1 Parallel Computing

In Chapter 3, I described Charlotte, a parallel computing environment for the Web. In Section 3.9, I presented several related projects, pointing out that most of these fail to provide the desired security guarantees. Besides Charlotte [7], other projects which address users' security concerns include Javelin [14], Bayanihan [54], and Ninflet [58].

These systems were specifically designed for parallel programming over the Web. By leveraging the ability of browsers to download and execute remote applets, they provide the means for any user, anywhere on the Web, using any Java-capable browser, to participate in a parallel computation. The essence of the

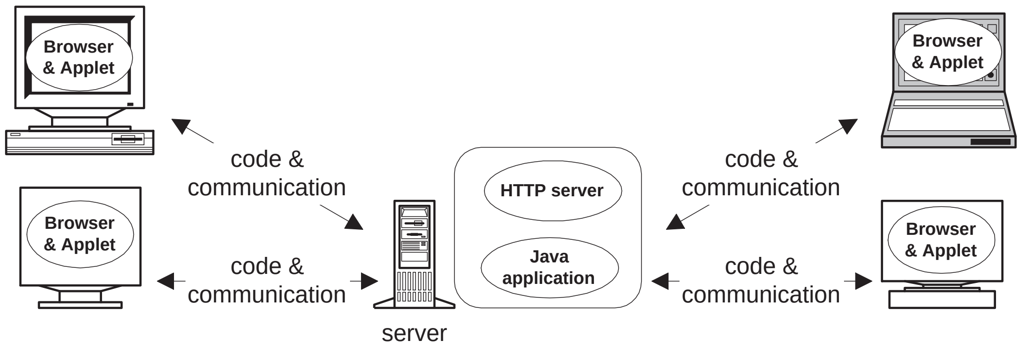


Figure 4.1: Architecture of typical Web-based parallel/collaborative systems.

architecture of systems such as Charlotte and Javelin is shown in Figure 4.1. A single host, the server in the diagram, runs a Java stand-alone application and an HTTP server. Other hosts load applets which reside on the server and execute them within a browser's secure environment.

There are two features common to the design of such systems. First, they require a host running an HTTP server in addition to a stand-alone Java application. In Charlotte, for example, the stand-alone application is called the *manager* and there is one manager per parallel program; in Javelin, it is called the *broker* and a broker may serve multiple programs at a time. The role of the stand-alone application is to distribute work among browsers and to handle traffic related to maintaining shared data. The second commonality is that users need to have *a priori* knowledge of the host running the stand-alone application.

4.2.2 Collaborative Applications

The second category of projects focuses on software infrastructures for collaborative applications on the Web. Examples of such applications are distributed whiteboards, calendars, and editors where multiple users collaborate towards one goal. High-level software systems which support collaborative Java applications include the Caltech Infosphere project [15], Jada/PageSpace [16], Java Collaborator Toolset [46], JavaSpaces [29], and TANGO [8].

Jada and JavaSpaces integrate the Linda concept with Java. Where Jada uses a Linda-like tuple model, JavaSpaces utilizes object hierarchies for tuple matching. PageSpace builds higher-level functionalities on top of Jada. Infosphere proposes channel-coupled agents to build flexible ways of collaborative interactions. These systems utilize Java for building stand-alone applications and do not take advantage of browsers. For example, while users interact with a PageSpace application in a browser, the actual collaboration is provided by a set of processes which must be started beforehand.

TANGO provides functionalities such as session management, user authentication, event logging, and communication between collaborative users. The main two elements of TANGO's architecture are a *local daemon* and a *central server*. The local daemon is implemented as a Netscape plug-in to circumvent restrictions put on applets, and hence, breaks the security guarantees. Java Collaborator Toolset provides a replacement for Java's standard Abstract Window Toolkit (AWT). The replacement package propagates AWT events to all collaborating agents.

While the functionality of these parallel computing and collaborative systems transcends KnittingFactory's goals, KnittingFactory can provide a flexible infrastructure for them. First, *KF Directory Service* can provide the mechanism for users' browsers to search the Web for work. And second, *KF Class Server* can be used to run initiators on machines other than a common HTTP server machine.

4.3 Design

The design of KnittingFactory has the following goals:

- *Ease of use:* KnittingFactory should work with standard browsers and comply to Java standards. In particular, users should not be burdened with additional administrative efforts such as locally installing software packages.
- *Secure environment:* The system security should not be jeopardized. This means that both native code and browser plug-ins should be avoided.
- *Scalability:* Since KnittingFactory targets computing over the World Wide Web, scalability is an important consideration.

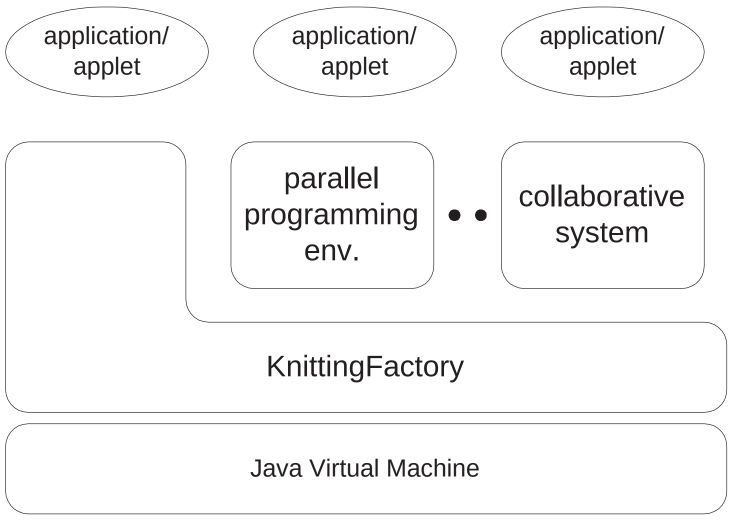


Figure 4.2: Layered design of KnittingFactory.

To take full advantage of today's technology, parts of the application designed to execute on remote machines are generally implemented as applets. The code to execute on a user's local machine is implemented as a stand-alone Java application which is not under scrutiny of the applet security model. For the rest of this Chapter, I will call the combination of a stand-alone application and applets a *distributed computation*, a particular invocation of this a *session*, an invocation of the stand-alone application an *initiator*, and applets belonging to the same session *partners*. This scenario is the setting for KnittingFactory.

While many of the systems surveyed in the previous Section provide high-level functionality to programmers and users, KnittingFactory is concerned with providing an infrastructure for such systems. Figure 4.2 shows the layered design of KnittingFactory. In Section 4.5, I will show how it can be used to support parallel programming environments; as an example, we will extend Charlotte's capabilities.

The design of KnittingFactory's services, *KF Directory Service* and *KF Class Server*, is outlined next.

4.3.1 Registration and Lookup

Distributed applications are generally easier to build and use when a lookup- or directory-service is available. While there exist many classical solutions for this (for example, CORBA name servers) our focus is to provide a light-weight solution for the Web. In particular, *KF Directory Service* has the following goals:

- Allow lookups from within a browser.
- Use the Web-infrastructure as much as possible.
- Avoid new daemon processes.
- Accommodate highly dynamic registration and deregistration of entries to support short-lived sessions.

There is a popular directory service called Lightweight Directory Access Protocol (LDAP) [62]. It specifies both a global data model and a protocol for clients and servers to access and maintain this directory. The servers perform search requests on clients' behalf. Our goal is to accommodate a large number of client requests for highly dynamic directory entries. The dynamics of the directories makes replication of information among distributed directory servers unattractive due to frequent invalidation. But without replication, the servers would have to perform considerable work to service the requests for possibly a large number of clients. Therefore, it is not feasible to perform the search within the servers, and consequently, LDAP-based directories are ill-suited for this purpose. We chose to pursue a solution which executes the search at the client side, within a client's browser.

Using dynamically loaded applets to realize a directory service is not a feasible approach. Consider an applet sent from host A and executing within a browser on host B. The applet can establish a network connection only to host A. If the user at B is interested in performing a directory service lookup, it is the case that (1) host A acts as a single central directory service for this and all other sites, or (2) host A performs the search on behalf of B, or (3) the directory information is replicated and synchronized on multiple sites—neither option is acceptable.

In KnittingFactory, we build upon existing HTTP servers to act as *directory servers*. Such a directory:

- stores information about sessions looking for prospective partners (such as the URL of the initiator, comments, etc.), and

- maintains a list of other *KF Directory Services*, which can be interpreted as the edges of a directed graph of *KF Directory Services*. Adding and removing both sessions and directories can be achieved using standard protocols (HTTP Post and CGI scripts).

The technique used in KnittingFactory to support lookup operations is unique. A lookup can be initiated at any server which offers a *KF Directory Service*. By including a JavaScript program along with a session and directory list in a single HTML page retrieved from such a server, a browser can inspect this information and, if a matching session is found, proceed to the respective URL; otherwise, it can use the directory list to search recursively at another site. By passing along state information as part of the URL of the next directory, search strategies like breadth-first search can be implemented. Breadth-first search is preferable search method to find sessions on nearby hosts first, assuming that nearby applications exhibit better communication efficiency than more remote ones.

The combination of these techniques allows the search to be performed in the client's browser, putting only minimal strain on directory servers. In Section 4.4, I will discuss the *KF Directory Service* in detail.

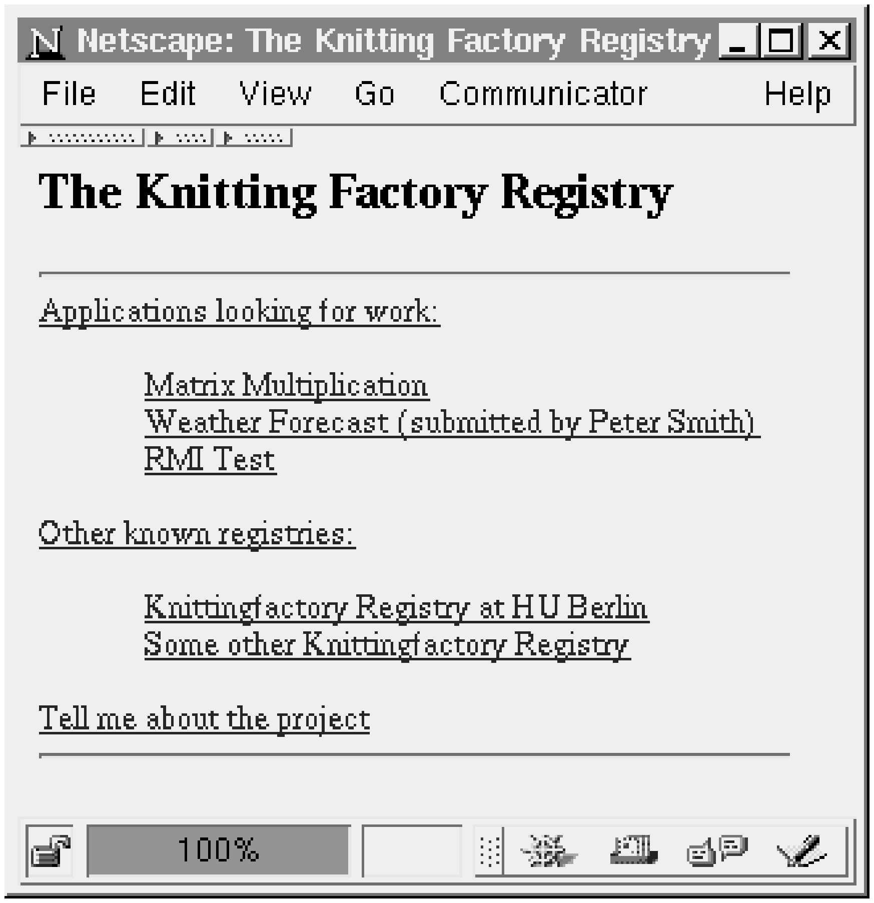
4.3.2 Arbitrary Origin of Applets

In a typical distributed Java application, applets are loaded from an HTTP server and connect to a stand-alone application, the initiator (see Figure 4.1). Among others, this is the case for applications written in Charlotte, Javelin, and many collaborative applications. The Java security model dictates that the initiator has to run on the same host as the HTTP server. Ideally, however, initiators should be able to execute anywhere on a network.

KnittingFactory addresses this problem by providing a light-weight HTTP server. *KF Class Server* implements the essential functionalities needed to serve applets, and is designed to be embedded into any Java application. As a result, it is the application itself which dynamically serves classes to browsers upon request. Although quite simple, this makes it possible for initiators to run on any host connected to the network. The details of *KF Class Server* will be discussed in Section 4.5.

4.4 Directory Service

KF Directory Service is implemented by a set of HTTP servers acting as directory servers, and a single HTML file at each server. Such a file contains a table of directory entries and a JavaScript program. While every HTML file contains the same JavaScript program, their directory entries may differ. A directory entry is



The Knitting Factory Registry

Applications looking for work:

Matrix Multiplication
Weather Forecast (submitted by Peter Smith)
RMI Test

Other known registries:

Knittingfactory Registry at HU Berlin
Some other Knittingfactory Registry

Tell me about the project

Figure 4.3: The main page of a KnittingFactory registry.

a tuple containing a string (i.e., the session description), a URL address (i.e., the initiator of the session), and optional fields for category, password, and comments. The category field is used to distinguish between entries representing sessions and other directory servers.

In the remainder of this Section, I will first describe registration details, and then discuss how lookups are performed.

4.4.1 Registration

Adding and removing an entry is performed by processes sending HTTP Post messages to HTTP servers, resulting in the execution of appropriate cgi-bin scripts located at each server. Each registry has a total of four cgi-bin scripts written in Perl which update the local HTML file. See Figure 4.3 for a sample HTML file containing three applications and two neighboring directories. The scripts used to update such a file are:

Adding a registry

newurl: The URL of the HTML page of a new registry. This adds a link pointing from the registry where this form is submitted to the given URL.

password: to secure consistency of the registry (optional).

Removing a registry

remurl Remove the given URL from the list of registries on this registry.

password: to secure consistency of the registry (optional).

Registering an application

host Name of the host where the application is running.

kfport The port number at which this application can be contacted by a Web browser to obtain an applet.

description An arbitrary description of this application (optional).

Deregistering an application

host Name of the host where the application is running.

kfport The port number, as used to register the application.

KnittingFactory provides a set of HTML pages to serve as a simple user interface to these scripts. These allow users to manually register and deregister applications and registries. Alternatively, applications can use two methods, `register()` and `deregister()` to register and deregister themselves at one or more registries. These methods, along with the class which implements them, will be described in Section 4.5.

4.4.2 Lookup

A *lookup operation* consists of loading an initial HTML page and executing the embedded JavaScript program with the request as a parameter. This JavaScript program analyzes the entries in its own page. If no match is found, a new URL is constructed and the browser is instructed to load the new address. The new URL contains the address of another directory server augmented with state information containing a list of hosts visited so far, and a list of hosts to visit next which is included in the `tag` part of the new URL. This part of the URL is usually used to let a browser navigate to a certain *anchor* in an HTML document. Since the HTML files at the registries do not contain any anchors, the browser just ignores the tag; but this information is still available to the JavaScript program.

The tag is organized as follows:

```
TAG ::= "search" + TODO-LIST + SEEN-LIST

TODO-LIST ::= ϵ |

              TODO-LIST-P

TOTO-LIST-P ::= URL |

                URL#TODO-LIST-P

SEEN-LIST ::= ϵ  |

              SEEN-LIST-P

SEEN-LIST-P ::= URL |

                URL#SEEN-LIST-P
```

where ϵ is the empty string and URL represents any HTTP address.

The script first checks if the tag actually starts with the string `search`. If this is not the case, it stops immediately. If it does, it first examines the links on this page to find out if any applications are registered with this registry. An application is distinguished from a registry by their relative position on the page. If applications are found, one is randomly selected and the browser is directed to this URL. It is possible to extend this to include additional parameters in the selection process, e.g., based on a best fit or best price principle.

If no matching application is found on a given page, the script will proceed with a breadth-first search. First, the URL of this page is appended to the `seen`-list of the tag string. Next, all registries contained in this page are appended to the `todo`-list. The first element of the `todo`-list is extracted used as the address of the next page to visit. To this address, the `#search` tag, the new `todo`- and `seen`-lists are appended to form the new URL. Finally, the browser is instructed to go to this newly constructed URL. Since this URL represents some other registry, it also contains the same script which will proceed with the search using the information obtained from the previous page and included in the tag. This process ends if either an application is located or no new registries can be found, in which case a suitable message is presented to the user or the calling application.

In summary, this approach allows the search to be performed by a JavaScript program without breaking the security restrictions imposed by Web browsers. Under the *tainting* [25] security model, a JavaScript program is not allowed to inspect pages coming from another site than itself. Once the JavaScript program finds an entry matching the request, the browser is instructed to follow the corresponding link. In this case, it will download an HTML page with an embedded applet which constitutes a partner of a distributed computation.

Note that this is not a search engine: unlike centralized search engines, the *KF Directory Service* is truly decentralized. The only services required from the HTTP server are sending the HTML file and running the cgi-bin scripts to add and remove entries. The actual search takes place within users' browsers.

4.4.3 Directory Service and Java RMI

While *KF Directory Service* was initially designed to work within browsers, in this Section, we will consider an example of how its services can be used in other contexts. In particular, we will look at the client/server architecture of Java Remote Method Invocation.

In Java RMI terminology, a *registry* is a remote object which provides basic name server functionality. The `rmiregistry` provided in JDK 1.1 is a shell script which invokes `RegistryImpl`, an implementation of such a registry. Two methods provided by the registry are of special interest to us: `bind()` and `lookup()` for registering and locating a server object, respectively. The existing RMI registry successfully binds a server object *only if it is local to its machine*. In the absence of a network-wide directory service, this means either that client applications must have *a priori* knowledge of the host running each server they might contact, or that all servers must run on a single well-known host.

KnittingFactory can overcome this restriction by providing a directory service to RMI registries. This requires the implementation of a wrapper class around `RegistryImpl`. In addition to passing `bind` operations to the RMI registry, the wrapper class makes a corresponding entry in a *KF Directory Service* under a specified service name. The lookup operation uses *KF Directory Service* to first search the network for the host running an RMI registry with the appropriate entry, and then contacts that registry for a server's remote reference. Thus, with the help of *KF Directory Service*, RMI servers can run and execute anywhere on the network, and clients can transparently get servers' remote references without having to know which hosts to search. The necessary wrapper classes will be realized in future versions of the system.

4.5 Class Server

KF Class Server provides the core functionalities needed to allow initiators to execute anywhere on a network, independent of whether a local HTTP is available or not. In particular, it serves an initial HTML page containing an applet tag, which implicitly results in additional requests for the applet classes which are served as needed.

4.5.1 Implementation

KF Class Server is implemented by the `KF_Manager` class. Any stand-alone application, such as an initiator, can instantiate a `KF_Manager` object to use its services. `KF_Manager`'s constructor takes the name of the applet class (i.e., partner) to be included in the initial HTML page and, optionally, the port number at which the application is expecting partners to contact it. `KF_Manager` itself creates a server socket on which it expects browsers to connect it and download the applet. Since our goal is only to serve one particular applet to the browser, an HTML file containing the applet name and optional parameters is generated and sent back to the browser. To initiate and execute the applet, the browser will connect again to the `KF_Manager` and request the applet. A request for a Java applet class results in asking the underlying Java Virtual Machine for the requested class. The class file is searched along the `CLASSPATH` environment variable and the byte-code is sent to the requesting browser. This process is repeated for every requested class.

Although any applet can be used this way, KnittingFactory provides the convenience class `KF_Applet` which extends Java's `Applet` class. This class provides a method `getPort()` which returns the port number where the application expects the applet to contact it. This means the actual applet would be derived from `KF_Applet` instead of `Applet`; it has to call the `init()` function of its parent class to allow correct initialization.

The integration with the directory service of KnittingFactory is achieved by two methods, `register()` and `deregister()`, both methods of `KF_Manager`. Both methods take one or more registry URLs as parameters and register or deregister the application at the respective registries.[1] The method `register()` automatically inserts the port number at which applets can be requested in the registration message. Thereby, a browser which has found an application via the searcher script will contact the application at the specified port and download the applet without further manual intervention.

KnittingFactory also provides an implementation of `KF_Manager` which integrates *KF Directory Service* to automatically register and deregister at one of more registries. In what follows, this implementation is used as an underlying layer for Charlotte.

4.5.2 KnittingFactory and Charlotte

In Chapter 3, I described Charlotte, a parallel computing environment for the Web. The two main components in a Charlotte program are: a *manager* (i.e., initiator) and one or more *workers* (i.e., partners). The manager process creates an entry in a well-known HTML page at initialization. Users can load and execute the worker code by directing their browsers to this URL.

[1]`register()` also takes an optional description parameter.

Charlotte requires that a manager runs on a host with an HTTP server to enable communication between a manager and the workers. With only six lines of code, KnittingFactory permits such a manager to be started on any host and to register itself with a set of *KF Directory Services*, allowing clients to find Charlotte applications easily. If only one machine with an HTTP server is available (a common setup), multiple Charlotte managers no longer need to run on this single machine, but they can be started on arbitrary machines.

4.6 Conclusions

In this Chapter, I have presented KnittingFactory, an infrastructure which supports building Web-based parallel and collaborative applications. Within the context of current research efforts, I pointed out a set of problems and challenges which application programmers face when using Java in combination with Web browsers and their ability to load and execute untrusted applets in a secure fashion. I described the two services provided by KnittingFactory addressing these issues: *KF Directory Service* which makes it possible for users with Web browsers to find a distributed session; and *KF Class Server* which allows users to start a distributed session on any host. I also argued how parallel programming and collaborative systems can benefit from using KnittingFactory as an underlying layer. This was demonstrated by utilizing KnittingFactory to extend Charlotte, a parallel programming environment.

Chapter 5

Conclusions

The Web represents a challenging environment for distributed computing because of the large number of heterogenous machines with different speeds which crash-fail and join arbitrarily, and networks with different performance characteristics which partition and merge. I have demonstrated that fault tolerance, fault masking, and scalability play an important role in Web-based systems. I argued that these issues appear in many areas and proposed new solutions.

I presented a decentralized design for allocating geographically dispersed, replicated Web servers and argued that this approach scales well. It uses pricing strategies which provide incentives to clients to influence the way they make dispatching decisions. An adaptive algorithm updates prices to deal with dynamic changes. Performance results based on a prototype were presented which validate its functionality.

I described a prototype system which allows programmers to write parallel applications in Java and allows anyone with a Java-capable browser to participate in a parallel computation. It comprises of a virtual machine model which isolates the program from the execution environment, and a runtime system which realizes this model on the Web. Two integrated techniques, eager scheduling and TIES, provide load balancing and fault masking transparently.

I presented an infrastructure for building applications composed of Java applets and designed to run within applets. It provides a distributed name service to assist users in finding networked applications on unknown hosts. Furthermore, its embedded class server eliminates the need for external HTTP servers.

Bibliography

[1] D. Andresen, T. Yang, V. Holmedahl, and O. H. Ibarra. SWEB: Towards a Scalable World Wide Web Server on Multicomputers. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, 1996.

[2] Y. Aumann, Z. M. Kedem, K. Palem, and M. Rabin. Highly Efficient Asynchronous Execution of Large-Grained Parallel Programs. In *Proceedings of the 34th IEEE Annual Symposium on Foundations of Computer Science*, 1993.

[3] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the 7th ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, 1996.

[4] A. Baratloo, P. Dasgupta, and Z. M. Kedem. Calypso: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. In *Proceedings of the IEEE International Symposium on High-Performance Distributed Computing*, 1995.

[5] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. KnittingFactory: An Infrastructure for Distributed Web Applications. Technical Report TR1997-748, Department of Computer Science, New York University, November 1997.

[6] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. Network Computing with Java Applets. To appear in *Concurrency: Practice and Experience*, 1998.

[7] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.

[8] L. Beca, G. Cheng, G. C. Fox, T. Jurga, K. Olszewski, M. Podgorny, P. Sokolwski, and K. Walczak. Web Technologies for Collaborative Visualization and Simulation. Technical Report SCCS-786, Northeast Parallel Architectures Center, January 1997.

[9] T. Berners-Lee, R. Fielding, and H. Nielsen. RFC 1945: Hypertext Transfer Protocol – HTTP/1.0, May 1996.

[10] N. Biggs. *Interaction models: Course given at Royal Hollaway College, University of London.* Cambridge University Press, 1977.

[11] R. D. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, A. Shaw, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Symposium on Principals and Practice of Parallel Programming*, 1995.

[12] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *Proceedings of the 7th ACM SIGOPS European Workshop*, 1996.

[13] T. Brisco. RFC 1794: DNS Support for Load Balancing, April 1995.

[14] P. Cappello, B. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 1997.

[15] K. M. Chandy, A. Rifkin, P. A. G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A World-Wide Distributed System Using Java and the Internet. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, 1996.

[16] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Redesigning the Web: From Passive Pages to Coordinated Agents in PageSpaces. In *Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems*, 1997.

[17] K. C. Claffy, H. W. Braun, and G. C. Polyzos. Tracking Long-term Growth of the NSFNET. *Communications of the ACM*, 37(8), August 1994.

[18] IBM Corporation. *Interactive Network Dispatcher User's Guide*, 1997. Available at http://www.ics. raleigh.ibm.com/netdispatch/ND2MST.HTM.

[19] O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y. M. Wang. One-IP: Techniques for Hosting a Service on a Cluster of Machines. In *Proceedings of the Sixth International World Wide Web Conference*, 1997.

[20] P. Dasgupta, Z. M. Kedem, and M. Rabin. Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. In *Proceedings of 15th International Conference on Distributed Computing Systems*, 1995.

[21] D. Dean, E. Felten, and D. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[22] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Server. In *Digest of Papers. COMPCON '96. Technologies for the Information Superhighway*, 1996.

[23] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar. A Novel Server Selection Technique for Improving the Response Time of a Replicated Service. In *Proceedings of the IEEE INFOCOMM*, 1998.

[24] A. Ferrari. JPVM – The Java Parallel Virtual Machine. http://www.cs.virginia.edu/~ajf2j/jpvm.html.

[25] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., 1997.

[26] I. Foster and S. Tuecke. Enabling Technologies for Web-Based Ubiquitous Supercomputing. In *Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing*, 1996.

[27] G. Fox and W. Furmanski. Towards Web/Java Based High Perfomance Distributed Computing – An Evolving Virtual Machine. In *Proceedings of the Symposium on High Performance Distributed Computing*, 1996.

[28] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing-Interface.* MIT Press, 1994.

[29] R. Guth. Sun's JavaSpaces is the Foundation for Future Distributed Systems, 1997. Available at http://www.javaworld.com/javaworld/jw-09-1997/jw-09-idgns.javaspaces.html.

[30] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[31] S. Hummel, T. Ngo, and H. Srinivasan. SPMD Programming in Java. *Concurrency: Practice and Experience*, 1997.

[32] Matrix Information and Directory Services Inc. *MIDS Internet Weather Report*. Available at http://www3.mids.org/weather.

[33] J. Ju and Y. Wang. Scheduling PVM Tasks. *Operating Systems Review*, July 1996.

[34] M. Karaul, Y. A. Korilis, and A. Orda. WebSeAl: Web Server Allocation. Technical Report TR1997-752, Department of Computer Science, New York University, December 1997.

[35] H. Karl. Bridging the Gap between Distributed Shared Memory and Message Passing. To appear in *Concurrency: Practice and Experience*, 1998.

[36] E. D. Katz, M. Butler, and R. McGrath. A Scalable HTTP Server: The NCSA Prototype. *Computer Networks and ISDN Systems*, 27(2), 1994.

[37] Z. M. Kedem and K. Palem. Transformations for the Automatic Derivation of Resilient Parallel Programs. In *Proceedings of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1992.

[38] Z. M. Kedem, K. Palem, M. Rabin, and A. Raghunathan. Efficient program transformations for resilient parallel computation via randomization. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, 1992.

[39] Z. M. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Combining Tentative and Definitive Algorithms for Very Fast Dependeble Parallel Computing. In *Proceedings of the 23rd ACM Symposium on Theory of Computing*, 1991.

[40] Z. M. Kedem, K. Palem, and P. Spirakis. Efficient Robust Parallel Computations. In *Proceedings of the 22nd ACM Symposium on Theory of Computing*, 1990.

[41] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, 1991.

[42] Y. A. Korilis, A. A. Lazar, and A. Orda. Architecting Noncooperative Networks. *IEEE Journal on Selected Areas in Communications*, 13(7), September 1995.

[43] Y. A. Korilis, T. A. Varvarigou, and S. R. Ahuja. Pricing Mechanisms for Distributed Resource Management. Technical Memorandum BL0112570-120396-TM3, Bell Laboratories, Lucent Technologies, December 1996.

[44] Y. A. Korilis, T. A. Varvarigou, and S. R. Ahuja. Optimal Pricing Strategies in Noncooperative Networks. In *Proceedings of the 5th International Conference on Telecommunication Systems: Modeling and Analysis*, 1997.

[45] Y. A. Korilis, T. A. Varvarigou, and S. R. Ahuja. Pricing Noncooperative Networks, June 1997. Submitted to *IEEE/ACM Transactions on Networking*. Available at http://www.multimedia.bell-labs.com/people/yannisk/price.html.

[46] B. Kvande. The Java Collaborator Toolset, a Collaborator Platform for the Java Environment. Master's thesis, Department of Computer Science, Old Dominion University, August 1996.

[47] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing*, 1988.

[48] D. Mosberger. Memory Consistency Models. Technical Report TR92/11, University of Arizona, 1992.

[49] Brown N. and C. Kindell. *Distributed Component Object Model Protocol – DCOM/1.0*, 1996.

[50] http://www.ncsa.uiuc.edu.

[51] http://www.netscape.com.

[52] M. Philippsen and M. Zenger. JavaParty – Transparent Remote Objects in Java. In *Proceeding of the ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation*, 1997.

[53] J. Pruyne and M. Livny. Parallel Processing on Dynamic Resources with CARMI. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, 1995.

[54] L. F. G. Sarmenta, S. Hirano, and S. A. Ward. Towards Bayanihan: Building

an Extensible Framework for Volunteer Computing Using Java. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[55] A. Siegel, K. Birman, and K. Marzullo. Deceit: A Flexible Distributed File System. In *Proceedings of the 1990 Summer USENIX Conference*, 1990.

[56] J. Siegel. *CORBA Fundementals and Programming.* Wiley, 1997.

[57] V. Sunderam, G. Geist, J. Dongarra, and R. Manchek. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 1994.

[58] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Ninflet: A Migratable Parallel Objects Framework using Java. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[59] S. Taylor. Prototype Java-MPI Package. Available at http://cisr.anu.edu.au/sam/java/java_mpi_prototype.html.

[60] http://www.top100.com.

[61] http://www.tucows.com.

[62] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). RFC 2251, December 1997.

[63] C. Yoshikawa, B. Chun, P. Eastham, Vahdat A., T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. In *Proceedings of the 1997 USENIX Annual Technical Conference*, 1997.

[64] M. Zekauska, W. Sawdon, and B. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proceedings of the Symposium on OSDI*, 1994.

Metacomputing and Resource Allocation on the World Wide Web

by

Mehmet Karaul

Advisor: Zvi M. Kedem

The World Wide Web is a challenging environment for distributed computing due to its sheer size and the heterogeneity and unreliability of machines and networks. Therefore, scalability, load balancing, and fault masking play an important role for Web-based systems. In this dissertation, I present novel mechanisms for resource allocation and parallel computing on the Web addressing these issues.

Large Web sites rely on a set of geographically dispersed replicated servers among which client requests should be appropriately allocated. I present a scalable decentralized design, which pushes the allocation functionality onto the clients. At its core lies a pricing strategy that provides incentives to clients to control the dispatching of requests while still allowing clients to take advantage of geographic proximity. An adaptive algorithm updates prices to deal with dynamic changes. A prototype system based on this architecture has been implemented and its functionality validated through a series of experiments.

Parallel computing on local area networks is based on a variety of mechanisms targeting the properties of this environment. However, these mechanisms do not effectively extend to wide area networks due to issues such as heterogeneity, security, and administrative boundaries. I present a prototype system which allows application programmers to write parallel programs in Java and allows Java-capable browsers to execute parallel tasks. It comprises a virtual machine model which isolates the program from the execution environment, and a runtime system realizing this machine on the Web. Load balancing and fault masking are transparently provided by the runtime system.